AN INCREMENTAL PROTOTYPING METHODOLOGY FOR DISTRIBUTED SYSTEMS BASED ON FORMAL SPECIFICATIONS

THÈSE N^{O} 1664 (1997)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Geir Jarle HULAAS

Ingénieur informaticien diplômé EPF

originaire de Lausanne (VD)

acceptée sur proposition du jury:

Prof. A. Strohmeier, directeur de thèse
Dr D. Buchs, corapporteur
Prof. P. Estraillier, corapporteur
Dr R. Guerraoui, corapporteur
Prof. B. Hirsbrunner, corapporteur
Prof. Ch. Rapin, corapporteur

Lausanne, EPFL
1997

There are indeed two labyrinths of the human mind:
One concerns the composition of continuum,
the other the nature of freedom;
and both take their source at the very infinity.

De Libertate, Gottfried Wilhelm Leibniz (1646-1716)

To Sandra

Acknowledgements

I am especially grateful to Professor **Alfred Strohmeier** who accepted the responsibility of being my advisor; I particularly appreciated his concern to ensure the best possible conditions for my research. It was a pleasure to be supported by his constant enthusiasm.

According to Knuth, natural computer scientists are « individuals who can rapidly change levels of abstraction, simultaneously seeing things “in the large” and “in the small” ». My scientific supervisor, Dr. **Didier Buchs**, positively has this quality. Didier provided me with the right balance of guidance and freedom to fulfill this work. His friendliness undoubtedly contributed to the success of this undertaking.

I would like to thank Professor **Pascal Estraillier**, Dr. **Rachid Guerraoui**, Professor **Beat Hirsbrunner** and Professor **Charles Rapin** for their effort to criticize my work, which was often only partly overlapping their respective research areas.

My sincere thanks to Professor **Charles Rapin** who also gave me the opportunity to work in his team and to make my first steps as a teacher and researcher.

I am indebted to Professor **Jean-Daniel Nicoud** for setting me in contact with Didier when I was searching for a subject in accordance with my interests and which could lead to a thesis.

The work presented here has depended a lot on my colleagues and friends from Didier’s *ConForM* group. In particular, I would like to mention **Pascal Racloz** for patiently reading and listening to my (sometimes chaotic) ideas, and **Nicolas Guelfi** for providing his high-level view of the semantics of CO-OPN. **Mathieu Buffo** certainly did a great job for the implementation of the SANDS environment. Cécile, Giovanna, Julie, Christophe, Jacques, Olivier and Stéphane also contributed to the friendly atmosphere in the group.

Finally I would like to thank my wife **Sandra** to whom I dedicate this thesis and the members of my family who encouraged me to pursue these long studies.

This work was funded by the Swiss Federal Institute of Technology of Lausanne, the European Esprit Long Term Research Project 20072 “Design for Validation” *(DeVa)* through the Swiss OFES (Office Fédéral pour l’Education et la Science), and the Swiss National Science Foundation project 2000.40583.94 “Formal Methods for Concurrency”.

Abstract

This thesis presents a new incremental prototyping methodology for formally specified distributed systems. The objective of this methodology is to fill the gap which currently exists between the phase where a specification is simulated, generally using some sequential logical inference tool, and the phase where the modeled system has a reliable, efficient and maintainable distributed implementation in a main-stream object-oriented programming language. This objective is realized by application of a methodology we call *Mixed Prototyping with Object-Orientation* (in short: *OOMP*). This is an extension of an existing approach, namely *Mixed Prototyping*, that we have adapted to the object-oriented paradigm, of which we exploit the flexibility and inherent capability of modeling abstract entities.

The OOMP process proceeds as follows. First, the source specifications are automatically translated into a class-based object-oriented language, thus providing a portable and high-level initial implementation. The generated class hierarchy is designed so that the developer may independently derive new sub-classes in order to make the prototype more efficient or to add functionalities that could not be specified with the given formalism. This prototyping process is performed incrementally in order to safely validate the modifications against the semantics of the specification. The resulting prototype can finally be considered as the end-user implementation of the specified software.

The originality of our approach is that we exploit object-oriented programming techniques in the implementation of formal specifications in order to gain flexibility in the development process. Simultaneously, the object paradigm gives the means to harness this newly acquired freedom by allowing automatic generation of test routines which verify the conformance of the hand-written code with respect to the specifications.

We demonstrate the generality of our prototyping scheme by applying it to a distributed collaborative diary program within the frame of CO-OPN (*Concurrent Object-Oriented Petri Nets*), a very powerful specification formalism which allows expressing concurrent and non-deterministic behaviours, and which provides structuring facilities such as modularity, encapsulation and genericity.

An important effort has also been accomplished in the development or adaptation of distributed algorithms for cooperative symbolic resolution. These algorithms are used in the runtime support of the generated CO-OPN prototypes.

Résumé

Cette thèse présente une nouvelle méthodologie de prototypage incrémental pour systèmes répartis spécifiés formellement. L'objectif de cette méthodologie est de combler le vide qui existe actuellement entre la phase où la spécification est simulée, généralement au moyen d'un outil d'inférence séquentiel, et la phase où le système modélisé possède une implémentation répartie, dans un langage de programmation à objets courant, et qui soit fiable, efficace et permettant la maintenance. Cet objectif est atteint par l'application d'une méthodologie que nous appelons le *prototypage mixte à objets* (sigle anglais: *OOMP*). Il s'agit d'une extension d'une approche existante, le *prototypage mixte*, que nous adaptons au paradigme objet dont nous exploitons la flexibilité et la capacité inhérente à modéliser des entités abstraites.

Le processus OOMP se déroule comme suit. Premièrement, les spécifications sources sont automatiquement traduites dans un langage à objets à structure de classes, fournissant ainsi une implémentation initiale portable et de haut niveau. La hiérarchie de classes générée est conçue de manière à laisser au développeur la liberté de dériver de façon indépendante de nouvelles sous-classes afin de rendre le prototype plus efficace ou pour ajouter des fonctionnalités qui ne peuvent pas être spécifiées dans le formalisme choisi. Ce processus de prototypage est poursuivi par petits incréments, ce qui permet de valider chaque étape par rapport à la sémantique de la spécification. Le prototype résultant peut finalement être considéré comme l'implémentation définitive du logiciel spécifié.

L'originalité de notre approche réside dans l'exploitation de techniques de programmation par objets pour l'implémentation de spécifications formelles afin d'augmenter la flexibilité dans le processus de développement. Simultanément, le paradigme objet donne aussi les moyens de maîtriser cette liberté nouvellement acquise en autorisant la génération automatique de routines de test qui vérifient la conformité du code écrit à la main par rapport aux spécifications.

Nous démontrons la généralité de notre méthode de prototypage par son application à un programme d'agenda collaboratif réparti; cet exercice est effectué dans le cadre de CO-OPN (*Concurrent Object-Oriented Petri Nets*), un formalisme de spécification très puissant qui permet d'exprimer des comportements concurrents et non-déterministes, et qui fournit de riches possibilités de structuration telles que modularité, encapsulation et généricité.

Un important travail a également été accompli pour développer ou adapter des algorithmes répartis pour la résolution symbolique coopérative. Ces algorithmes sont utilisés dans le support d'exécution des prototypes générés sur la base de spécifications CO-OPN.

Table of Contents

Chapter 1
Introduction

Chapter 2
An Incremental Prototyping Methodology based on Formal Specifications

Chapter 3
The CO-OPN Specification Language

Chapter 4
Operational Semantics of AADTs

Chapter 5
Prototyping of AADTs

Chapter 6
Operational Semantics of CO-OPN Objects

Chapter 7
Prototyping the Object Application Layer

Chapter 8
Conclusion

List of Figures

Chapter 2. An Incremental Prototyping Methodology based on Formal Specifications

Chapter 3. The CO-OPN Specification Language

Chapter 4. Operational Semantics of AADTs

Chapter 5. Prototyping of AADTs

Chapter 6. Operational Semantics of CO-OPN Objects

Chapter 7. Prototyping the Object Application Layer

Appendix A. Major Control Algorithms for CO-OPN

Appendix E. An Execution Cycle of the Collaborative Diary

Chapter 1

Introduction

1.1 Our Motivation

One of the most compelling objectives of Software Engineering is to reduce the development cost of computer programs. In a formal specification-based approach, automatic implementation processes can reveal themselves particularly profitable, not only through the reduced development time they may provide, but also because of their increased reliability compared to entirely manual coding methods.

The advantage of formal methods resides by definition in their unambiguous nature. They can thus be supported by tools which automatically verify the properties, extract useful information or transform them into other representations. Moreover, they require a thorough analysis of the modeled system during the very first levels of the development process. The benefits of formal specifications become even more evident when they can be used continuously through several stages of the software development cycle, like specification, implementation, integration and testing.

This ambition is however only partly fulfilled by existing methods, because the different development phases are poorly integrated in several ways. The reasons emanate mainly from unwanted paradigm shifts during the development cycle in the models, languages and tools used. The purpose of this thesis is to fill the gap which currently exists between the phase where a specification is simulated, generally using some logical inference tool, and the phase where the modeled system has an efficient and maintainable implementation in a main-stream object-oriented programming language. This objective is realized by application of a new methodology we have named *Mixed Prototyping with Object-Orientation* (OOMP): This is an extension of an existing approach, namely *Mixed Prototyping* [Choppy 87], which it adapts to the object paradigm in order to exploit the flexibility and inherent capability of modeling abstract entities.

1.2 An Interesting Initial Solution: Mixed Prototyping

The principle of mixed prototyping is to produce programs in a high-level programming language by compilation of formal specifications. The generated code is structured so as to allow, with the help of a tool called the *integrator*, its progressive replacement by more efficient or functionally more complete hand-written modules. A remarkable characteristic is that the prototype may be executed at any stage of the development, with a total cooperation between the modules of both environments: This is also sometimes referred to as *heterogeneous prototyping*.

Compared to previous approaches, mixed prototyping defines both a safer and more efficient frame for the developer. It is safer, because it relies on the use of a strongly typed programming language, Ada (in its initial version [Ada 83]), which limits the possibilities of inconsistency. It is more efficient, because the data types which are shared between the automatically generated and the hand-written environments do not need any run-time conversions. Also, the prototyping tool must no longer necessarily provide an extensive set of built-in data types and primitives to be attractive, since the user can himself easily supply equivalently powerful libraries.

Apart from the purely programmatic advantages quoted above, mixed prototyping has several positive implications on the methodological level [Choppy 87]:

- Mixed prototyping supports both top-down and bottom-up development strategies.
- It promotes systematic refinement schemes.
- There is no need for *stubs* or *driver routines* for interconnecting the automatically generated and the hand-written modules, since they are written in the same programming language and are designed to cooperate following a predefined protocol.
- The interfaces and the behaviour of the specification modules are well defined at the very beginning of the mixed prototyping process. The parts which have not yet received a concrete implementation may be entirely simulated by the automatically generated modules. Therefore the need for *integration tests* becomes almost insignificant.
- Reuse of concrete basic modules brings instant efficiency to new developments.

All these properties show that mixed prototyping is well suited for concurrent engineering, a methodology where the different parts of a product are developed simultaneously by several teams, and where incremental and heterogeneous prototyping are of great value.

1.3 Methodological Contributions

The proposal of this thesis is to adapt the technique of mixed prototyping in order to generate the code in an object-oriented programming language, e.g. the last version of Ada [Ada 95]. The result is a more natural and intuitive prototyping process since the intrinsic resources of the object paradigm, such as inheritance and dynamic binding, eliminate the need for an integrator tool. Our approach also provides additional flexibility, because it is less language-dependent, and in any case the implementation granularity is not the module, but the subprogram. Another important advantage is that the automatically generated code, which is certified correct with respect to the specifications, is not deleted when the developer integrates a piece of hand-written code. Therefore the method not only promotes code reuse, but also allows interesting combinations: It supports for instance a form of validation of the hand-written modules by using the automatically generated code as assertions which are tested at run-time. In all generality, the finer-grained incremental nature of the OOMP process implies that it is easier to detect and localize deviations of the hand-written code from its intended meaning. In other words, our extension of mixed prototyping gives better guarantees that the resulting piece of software conforms to its specifications.

Since *mixed prototyping with object-orientation* is a prototyping method, which subsumes a minimal level of responsiveness from the development environment, we have incorporated an additional code pattern in the generated prototypes, the purpose of which is to reduce the development time and make the interaction with the prototyping tool more comfortable: It is a form of *creational pattern* [Gamma et al 95], i.e. a structure which allows us to integrate new implementations of an abstract data type without having to edit and recompile all the client modules. The secret of this flexibility lies in an extensive usage of abstract classes and dynamic binding.

1.4 Technical Contributions

The concept of mixed prototyping was initially put into practice in the field of algebraic specifications of abstract data types. This thesis extends that work by illustrating how *OOMP* can similarly be exploited in the development of complex distributed software systems specified with the *CO-OPN* formalism [Buchs&Guelfi 91]. We have designed a compiler which automatically generates incremental prototypes for networks of distributed memory machines on the basis of CO-OPN specifications. We expose here the original techniques which were required to execute the prototypes in a distributed environment. The difficulties were twofold: First, the semantics of CO-OPN requires a lot of global knowledge about the system, which must be minimized in order not to overload the communica-

tion channels, and it also displays many levels of non-determinism, which implies the ability to restore previous states in a coordinated manner. The second difficulty was to respect the additional constraints imposed by the prototyping frame: The generated code must be readable, efficient and reliable. The combination of these non-trivial requirements is discussed in depth in this report.

Another contribution, from a purely technical point of view, is that we give here the first implementation scheme for *algebraic Petri nets* [Reisig 91]. This work was necessary since CO-OPN is a strict superset of this formalism.

1.5 Structure of the Report

The plan of this thesis is as follows. In the next chapter we will first introduce the existing notion of mixed prototyping [Choppy 87], and then our contribution, the incremental prototyping scheme which we call *OOMP*. This description will stay at a rather abstract level, in order to show its general usability. From there on, the rest of this report will be devoted to demonstrating the applicability of the methodology to the entirety of CO-OPN: This is an interesting challenge, since CO-OPN is a specification language which comprises both *algebraic abstract data types* (AADTs) [Ehrig&Mahr 85] and *algebraic Petri nets* (APN) [Reisig 91], organized as objects interconnected by synchronization links [Buchs&Guelfi 91]. This formalism also incorporates many structuring concepts which makes it suitable for specifying large systems. Therefore, we continue with chapter 3 which presents the CO-OPN language as it was designed by Buchs and Guelfi. Chapter 4 details the compilation algorithm for AADTs, which is in fact a slight adaptation of the paper [Schnoebelen 88]. Chapter 5 describes our new form of code generation in Ada95 and how OOMP may be applied to it. In chapter 6, which constitutes the more technical contribution of the thesis, we show how APN objects are compiled and describe the run-time support needed for the distributed execution of the prototypes. Thereafter, chapter 7 shows to which extent the OOMP process may be applied to CO-OPN objects. This latter part describes our innovative work to apply mixed prototyping to objects with both concurrent and non-deterministic behaviours. Finally, in the last chapter, we conclude the report with a synthesis of the problems encountered and the contributions of the thesis.

Chapter 2

An Incremental Prototyping Methodology based on Formal Specifications

2.1 Foreword

The objective of this chapter is to present the basic principles of our incremental prototyping methodology. We will stay at a rather abstract level, since we want to demonstrate the generality of the approach. It is only in the following chapters that we will show how the prototyping methodology applies to definite specification formalisms, which are, respectively algebraic abstract data types and the CO-OPN version of modular algebraic Petri nets [Buchs&Guelfi 91]. It may by the way be interesting to know that CO-OPN/2, the fully object-oriented incarnation of the language [Biberstein&Buchs 95], has been chosen as formalism for the European ESPRIT Project "Design for Validation" (DeVa).

The plan of this chapter is as follows: First we give an overview of the different existing specification-based prototyping techniques and their role in the software development cycle. Then we focus on the concept of *mixed prototyping*, explain how it is put into practice in terms of tools and programming language constructs. Thereafter we show the advantages brought to the previous work by the object-oriented paradigm, subject which constitutes the core of our contribution under the name of *mixed prototyping with object-orientation* (*OOMP*).

The last sections of this chapter discuss the approach of OOMP on different levels. We start by arguing about the validity of OOMP on the semantic plan, in order to demonstrate its correctness. Having done this, we briefly present the interconnection of the tools which are to support this methodology within the SANDS environment [Buchs et al 95]. Then we briefly mention some problems left open by this thesis, before reviewing and comparing with other existing incremental prototyping techniques.

2.2 Introduction

Whereas hardware is becoming increasingly cheaper and more reliable, making the application domains of computers ever more numerous, not only in everyday life, but also in more safety critical fields, software has not followed the same successful evolution. This is because software engineering, and computer science in general, is by nature completely different from other disciplines, due essentially to the fact that its constituents are immaterial. Computer science is developing the paradigms and methods for the exploration of the world of information and intellectual processes that are not directly governed by physical laws [Hartmanis 94]. This is what makes this branch so different from all other domains of science and engineering, and at the same time so difficult to master.

Software engineering may be defined as the discipline which deals with developing systematic methods for the creation of computer programs that are useful, reliable and cost effective. This concern is expressed by the notion of *software process*, which models the range of activities centered around the entire span of the software life-cycle (Figure 1).

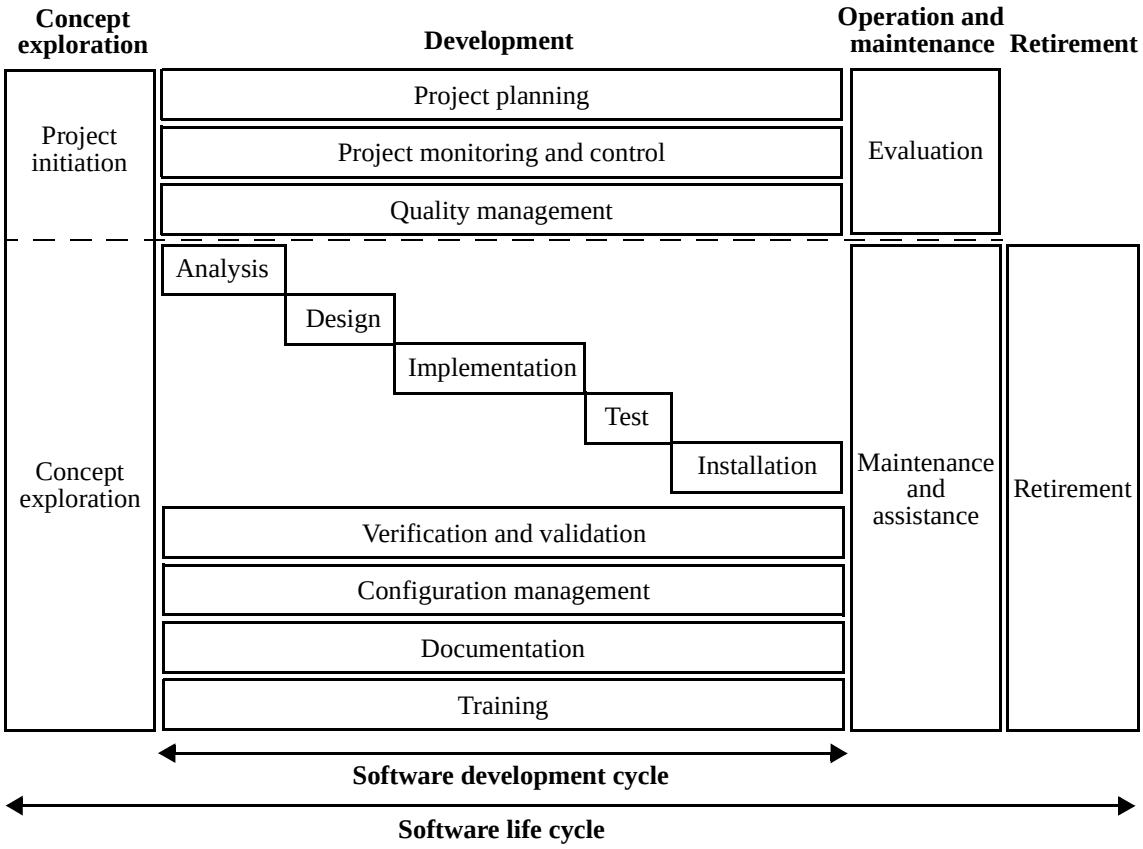


Figure 1. The Software Life Cycle

In this figure (from [Strohmeier 96]) the part over the dashed line represents project management tasks, while the lower portion shows the technical activities. The software life cycle illustrated here is the traditional “waterfall” model, which has many slight variations. Its specificity is that each phase of the development must be terminated according to precise criteria, and the completed results are then frozen, before the next phase can begin.

There is however a frequent complaint about this model, saying is that its structure is too rigid: It does not recognize the importance of iteration in the software process [Sommerville 92], a flaw which hinders efficient exploitation of prototyping and formal methods.

2.3 Prototyping in the Software Development Cycle

Prototyping can be applied with different objectives and at different levels of the development process, for instance when:

- Determining the feasibility of the product;
- Evaluating the realization cost of the development;
- Establishing and validating the client’s requirements;
- Validating the functional specifications;
- Needing estimations about the performance of the system, like for instance an evaluation of the execution costs;
- Using the prototype as a model, or a dynamic representation of it, especially during system testing.

Prototyping may be simply considered as beneficial for the general know-how which is acquired in building a given system.

Different Approaches to Prototyping

When starting from the initial requirements, [Hallmann 91] proposes a classification of prototyping methods into three main categories:

- *Exploratory* programming is the most primitive method. It is very close to traditional application development, in that there is no specification: Modules are added to the system one after another, the ones not yet implemented being simulated to allow testing. This approach may lead to unexpected results when integrating a new module because of its side-effects. This problem is particularly acute in parallel environments.
- *Throw-away* prototyping, where the prototype mainly serves as a way of validating the

requirements - or part of them - after which the operational system has to be constructed from scratch. This technique is to be used when the primary goal is rapid and flexible system evaluation, and when the prototyping environment lacks reliability or execution efficiency. In this category we differentiate *simulated* and *stand-alone* prototypes. In the former, execution is possible only inside the development environment, whereas in the latter, the resulting prototype is autonomous.

- *Incremental* prototyping, where the prototype is developed through various steps leading to the real end-user application. This implies that the prototyping tool provides for acceptable characteristics concerning completeness, efficiency and robustness. Within this category, [Asur&Hufnagel 93] identify an additional subset: Prototyping may be said *evolutive* as long as the developer still has some freedom at the level of the system architecture, by opposition to phases where the structure of the prototype is already freezed, permitting only modifications to the functional aspects.

Our proposal belongs to the third kind of prototyping technique, since it stems from the early stages of the software life cycle down to the complete implementation. We may however precise that it is a *combination* of evolutive and simply incremental prototyping methods, since the prototyping tool settles both the structure and the behaviour of the application or module. During the first stages of development, prototyping is applied on the level of the specifications: The automatically generated code is then only used for simulation purposes and thrown away when the result is unsatisfactory. This is the evolutive part of the prototyping process. When the developer decides that the specification has reached a sufficient level of maturity, he can start working on the level of the programming language and progressively replace the generated modules by more efficient or adequate hand-written code. This last phase is the incremental style of prototyping (of which mixed prototyping is a subset, as described later).

As a further classification, one can also distinguish different *functional* purposes for the prototype:

- In *vertical* prototyping only certain critical functions are represented. They are considered as independent from the rest of the system.
- In *horizontal* prototyping, all the functions of the final product are accounted for, but sometimes only at the level of a draft, leaving aside certain aspects that are not mandatory for the current aims.

One can easily combine these two approaches inside any given prototype, especially in prototyping environments like ours, where emphasis is set on modularity.

The Benefits of Formal Specifications

A rigorous method, which allows proving certain characteristics of the specified system, can only be formal, i.e. entirely based on mathematical concepts. The objective of the specification phase is to clearly state the set of functionalities awaited from the software. The ever increasing complexity of software systems imposes a progressive procedure based on abstraction, refinement and enrichment. To gain better control over this part of the development, it is preferable to have structuring primitives. The possibility of organizing a formalism or language in a modular way has become a necessary attribute for specifying substantially sized systems. The CO-OPN language supports additionally a notion known from traditional high-level languages, namely genericity. Moreover CO-OPN enforces hierarchy and is therefore well-suited for large-scale systems.

What are the benefits gained from prototyping of formal specifications? They are multiple, mainly:

- The formal approach provides security by its inherent unambiguous nature, and by the support of tools which can automatically verify properties, extract information and transform the specifications into other representations. Formal specifications force in-depth analysis at an early stage. This is important since the sooner an error is caught in the development process, the simpler - and less expensive - it is to remove.
- During the functional analysis, prototyping contributes to completing the specification of the product: The engineer compares the behaviour of the prototype with what he intended to specify. The feedback is immediate, since the prototype is the execution of the specifications.
- If the code generator of the specification compiler is certified correct, the prototypes produced may be considered as perfect implementations of the specifications. There is no risk of misinterpreting the specifications, versus when a human programmer has to produce the code manually from his understanding of the specifications.
- The prototype constitutes a model during the testing process, when it is not itself considered as the actual product.

Using formal specifications, this quantity of material is in fact obtained at very low expense: The cost of producing a prototype is negligible, since it simply constitutes the execution of the specification.

Refinement of Formal Specifications

Sommerville states that the only realistic way to construct a formal specification is to proceed incrementally [Sommerville 92] (see Figure 2).

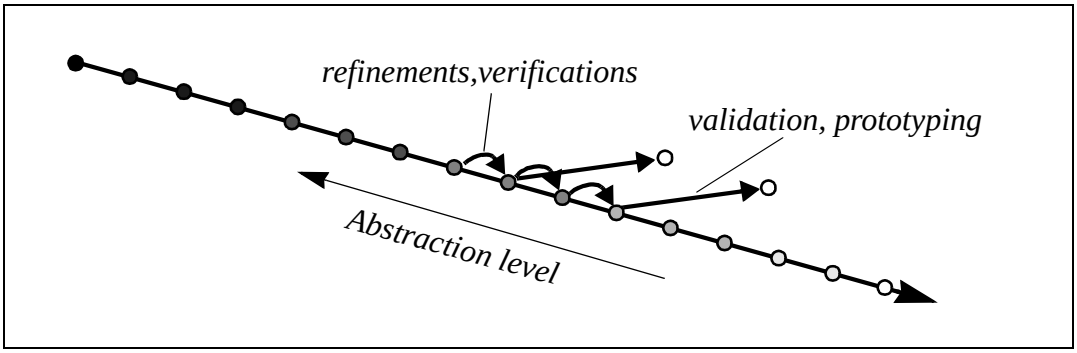


Figure 2. The Continuous Model

The initial perception of the system to build may be very vague, indeed. As analysis and simultaneous validation progress, vision of the architecture, algorithms and associated data structures improve until final implementation shape. An important aspect of refinement is the problem of preserving the semantics of the specification. Therefore it is advisable to proceed by small steps, which consist in easily provable modifications. This approach is called the *transformational development* technique, and is naturally connected with code generation tools which perform the translation of the most refined version of the formalism into an implementation.

2.4 Incremental and Heterogeneous Prototyping

Use of formal specifications becomes especially attractive when they can be exploited continuously through several stages of the software development cycle, like specification, implementation, integration and testing. This ambition is however only partly fulfilled by current methods, because the different development phases are poorly integrated in several ways. The reasons emanate mainly from unwanted paradigm shifts during the development cycle in the models, languages and tools used.

The question of deciding whether specifications should be executable or not is an old controversy (see e.g. [Gravell&Henderson 96]). A major problem is that there is a trade-off between the expressive power and the efficiency of implementation and validation of a specification formalism; a compromise has to be found [Hansson et al 90]. The position we will defend in this thesis is that the same language can (and should) be used for simulation during a wide range of levels of abstraction: The role of the specification formalism may span from the expression of the software requirements specification down to the detailed design stage. Then, a high-level programming language has to provide the mechanisms for a smooth translation from the formalism by keeping a similar level of abstraction. Finally,

prototyping at the programming language level, by exploitation of the object paradigm, allows specializing the code until a satisfactory implementation is found.

There seems at least to be a consensus about the fact that there is no universal formalism which fits any problem domain [Murphy et al 89]. In this work we use CO-OPN, which is a very complete specification language, but is for instance best suited for the description of discrete systems.

A salient difficulty is to compile the specifications into both efficient and useful programs. The specification compiler may generate a highly optimized and thus hardly legible code, which makes it inappropriate for manual modifications. The developer might however want to add functionalities, e.g. customized I/O or operations on irrational numbers, which often cannot be specified in usual formalisms. This dichotomy has been answered by two different approaches:

1. The usual solution is to apply transformations to the specifications in order to produce optimized code or to extract maximal parallelism. This is the approach of [Bréant&Pradat-Peyre 94] for instance, and has also been applied to CO-OPN, but upstream of the compilation phase [Buchs et al 96]. Then, in order to incorporate hand-written code, the most primitive approach is to generate program skeletons, where the developer is invited to fill empty subprograms which will be called automatically at run-time. More sophisticated solutions exist, where a brief description of the external components is included in the specification, which allows a semantically cleaner connection with the generated prototype. One of the goals in [Kordon 92] is to take into account pre-existing software components.

2. The second approach is to give the developer the possibility to progressively replace the automatically generated modules by hand-written code. The European IPTES project [IPTES 94] proposes a module interconnection protocol where the developer has to define conversion routines for his hand-written code to exchange data with the high-level specification simulator: This allows for instance the incorporation of components which are not judged critical enough to be completely specified in a formal way. A consequence of the IPTES approach is that only bottom-up implementation schemes are supported. On the other hand, *mixed prototyping* [Choppy 87], which we describe in the next section, does not require any run-time conversion and also leaves open the choice of the development strategy.

The conjoint development and execution of modules at different abstraction levels is sometimes referred to as *heterogeneous prototyping* [Gabriel 89]. An interesting aspect of heterogeneous prototyping is that it is well suited for *concurrent engineering*, a methodology where the different parts of a product are developed simultaneously by several teams. The objective of the IPTES Project was to experiment with Boehm's *spiral model* [Boehm 88]

for software development [Pulli&Elmstrøm 93]. Briefly, this model gives more flexibility than the traditional waterfall approach by focusing on risk management and teamwork. Notions such as incremental prototyping and concurrent engineering fit well into this model.

One inconvenience with the spiral model is however that it is very abstract. It is in fact a *meta-model* which must instantiated according to a wide range of factors, such as: level of understanding of the future product, technical ability of the development team, available technology, time and budget pressure, or even legal and environmental concerns.

2.5 Mixed Prototyping

Based on both incremental and heterogeneous prototyping, *mixed prototyping* is a technique which partly answers the desire of safety and continuity in the development. The principle of mixed prototyping is to produce programs in an imperative programming language by compilation of formal specifications. The generated code is structured so as to allow, with the help of a tool called the *integrator*, its progressive replacement by more efficient or functionally more complete hand-written modules. The automatically generated modules are said *abstract*, and the user's implementations are called *concrete*.

Figure 3 illustrates a hierarchy of modules under development where some modules are abstract, some completely concrete, and some others in an intermediate state, called *semi-abstract*. This situation still allows to execute the prototype, provided that it is in a minimally coherent state.

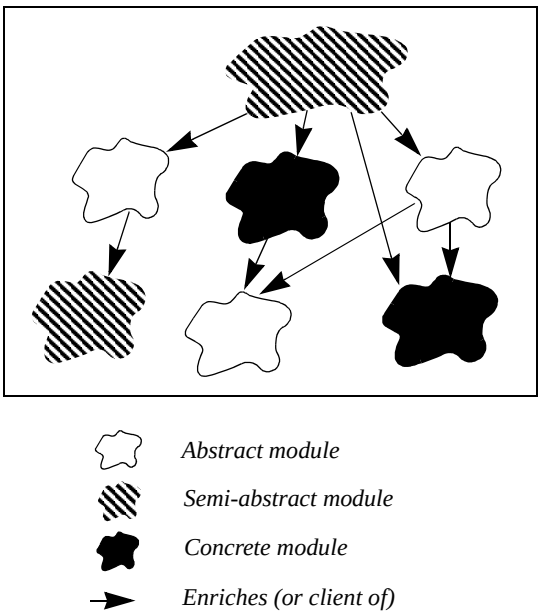


Figure 3. The Concept of Mixed Prototyping

A remarkable characteristic of mixed prototyping is thus that the prototype may be executed at any stage of the development, with a total cooperation between the modules of both environments.

Compared to previous approaches, mixed prototyping defines both a safer and more efficient frame for the developer. It is safer, because it relies on the use of a strongly typed programming language, Ada (in its initial version [Ada 83]), which limits the possibilities of inconsistencies. It is more efficient, because the data types which are shared between the automatically generated and the hand-written environments do not need any run-time conversions; this is easier to realize when the abstract and the concrete code are in the same programming language. Also, the prototyping tool must no longer necessarily provide an extensive set of built-in data types and primitives to be more attractive, since the user can himself easily supply equivalently powerful libraries.

Apart from the above-mentioned purely programmatic advantages, mixed prototyping has several positive implications on the methodological level [Choppy 87]:

- Mixed prototyping (in the version we use here, see section 5.5 on page 127 for other approaches) supports both top-down and bottom-up development strategies.
- It promotes systematic refinement schemes.
- There is no need for *stubs* nor *conversion routines* for simulating or interconnecting the automatically generated and the hand-written modules, since they are written in the same

programming language and are designed to cooperate following a predefined protocol.
- The interfaces and the behaviour of the specification modules are well defined at the very beginning of the mixed prototyping process. The parts which have not yet received a concrete implementation may be entirely simulated by the automatically generated modules. Therefore the need for *integration tests* becomes almost insignificant.
- Reuse of concrete basic modules brings instant efficiency to new developments.

The correctness of partial implementations of models, which this technique is based on, was formally proved in [Choppy&Kaplan 90]. The technique has been used exclusively in relation with algebraic specifications of abstract data types, although the authors have stated that it was not at all limited to this kind of formalism [Choppy et al 89].

Mixed prototyping has been experimented with several programming languages such as C or Lisp, but was essentially put into practice with the Ada language [Ada 83] because of its strong typing and support for genericity. In this context the role of the prototyping tool, *ASSPEGIQUE* (1) [Choppy 88], is to:

1. Provide dummy Ada function bodies for the operations which are not implemented in the concrete code. This is necessary for the generic instantiation to take place. Notice that the knowledge of which operations are concrete must either be directly fed by the user or by syntactical analysis of the user's code.
2. Generate some glue code, the role of which is to choose the concrete operations and data types when available, and otherwise to take the abstract code.
3. Compile the client modules because of the generic package reinstantiation.

At first sight it would seem that Ada's type derivation mechanism could have been used to simplify this integration of the user's definitions. The problem is that one cannot change the internal representation of a derived type in Ada83, as opposed to Ada95. In ASSPEGIQUE, type derivation is however used in some restricted cases to implement the notion of *coercion*, which is a kind of inheritance mechanism in algebraic specifications.

An immediate remark about the scheme described above is that the integrator is very present during the prototyping process and it is dependent on the target programming language, since it must regenerate glue code at each modification of the concrete code, as well as the required dummy function bodies. To be really user-friendly it should be able to perform syntactical analysis of the concrete modules. This technique can probably be slightly simplified when the implementation granularity is the module instead of the subprogram.

1. ASSPEGIQUE stands for "ASsistance à la SPécification alGébrIQUE".

2.6 Contributions of the Object-Oriented Paradigm

Our proposal is to adapt the technique of mixed prototyping in order to generate the code in an object-oriented programming language, e.g. the last version of Ada [Ada 95]. This seems rather natural since the object paradigm is an ideal support for the notion of abstract data type. We will basically use two mechanisms:

- Providing several implementations for a single abstraction. The abstraction is defined by the formal specifications, both syntactically, through the reference to a unique interface, and semantically, through the properties expressed by the axioms.
- Mixing a class-based and a prototype-based style of programming gives us the security constraints of the first and the flexibility of the latter.

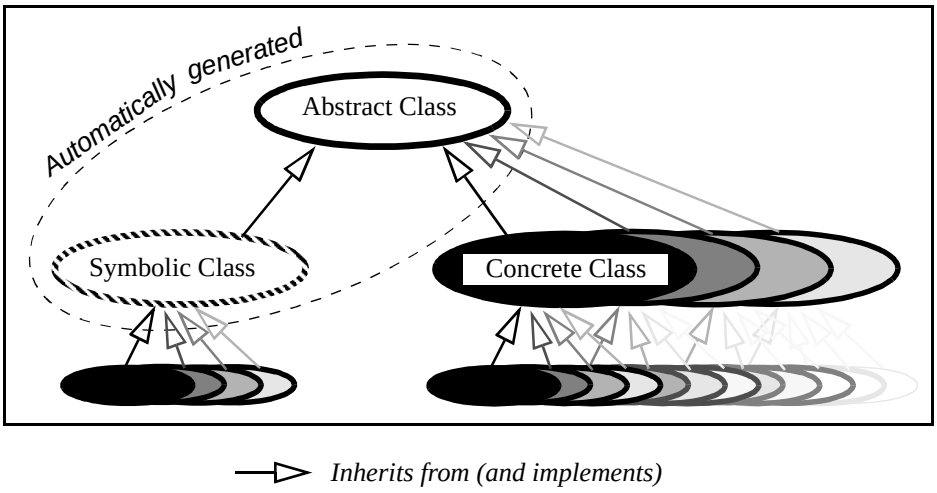


Figure 4. Object-Oriented View of Mixed Prototyping

2.6.1 A Class-Based Decomposition of Functionalities

We want to apply the results of mixed prototyping to the frame of object-oriented languages, because they offer a intrinsic paradigm, which is to allow several implementations for the same abstraction. To this purpose, we adapt our terminology to the usual object-oriented vocabulary: From now on we will talk of *abstract classes*, *symbolic classes*, and *concrete classes* (Figure 4 depicts this organization). The former two are generated by the compiler, and the latter category denotes everything which is hand-written. The abstract class provides a set of methods, implemented by compilation of the specifications. It has however no internal representation: That is the role of the symbolic class(2), which provides the minimal implementation required for the methods of the abstract class to work properly.

To make these statements a bit more concrete, we may say that each module of the specification formalism corresponds to a class hierarchy. In the case where the specification language is itself object-oriented, we will of course try to map each class construct of the source language to a class hierarchy of the target language. In general, if the specified entity consists in a type definition, such as a *sort* in algebraic specifications, then it may be advantageous to take that as central entity in the prototyping process. The choice of a mapping strategy is done once for all according to the nature of the specification formalism when the prototyping tool is designed.

Use of object-orientation adapts advantageously the principles of mixed prototyping to modern programming languages and paradigms. Compared to previous results, we can state the following additional improvements:

- The developer autonomously redefines and specializes operations of the abstract class. The prototyping tool gets less language dependent because it does not have to know what changes have been applied; it only needs the name of the class to be used as implementation. It becomes a pure code generator and is consequently easier to retarget to new programming languages.
- There is no hidden mechanism: We exploit the natural resources of the object-oriented paradigm. The prototyping process gains transparency.
- Memory management is transparent to the clients. The abstract class can even discharge its own descendant classes of this burden. See section 2.9 on this subject.
- The methods of the abstract classes, which are guaranteed correct w.r.t the specifications, are inherited by the concrete classes, and do no longer necessarily disappear when the developer wants to redefine them: The method permits code reuse.
- Since it knows the interfaces of the specifications, it is possible for the prototyping tool to automatically derive classes from the user's hand-written code which will be transparently incorporated and used by the client classes without their knowing. This opens the way to new techniques for tracing the execution and testing the correctness of the concrete code, as described in section 2.8.

We would like to emphasize that OOMP is based on well-established programming principles for data abstraction, see e.g. [Meyer 88]. The rules are that the code of the abstract classes must never directly access the internal structures, but instead express all operations in terms of *accessor*, *transformer* and *constructor* functions. The concrete classes are then related to their abstract class by an abstraction function.

2. The name *symbolic* comes from the fact that the automatically generated data structures will often be defined as in symbolic manipulation languages, i.e. by attaching to each object a tag which will be used as type identification at run-time.

What distinguishes OOMP from this classical approach is, of course, that part of the hierarchy is the result of compiling formal specifications, but also the following: In usual patterns we have one parent class with several equally important sibling implementations. We propose instead a hierarchy with one privileged implementation, the one generated automatically, which will serve as reference for ulterior hand-written class derivations.

2.6.2 The Flexibility of Prototype Objects

The most common application of object-orientation consists in designing and implementing reusable software components, and the modeling power it provides does not have to be proven anymore. Besides that, the notion of dynamic binding, which is inherent to this paradigm, has been extensively exploited because it contributes to data abstraction, and also because it provides more flexibility from a purely programmatic point of view.

For a prototyping environment to be comfortable, it must show a certain level of responsiveness: The activity of prototyping is inherently connected to the notion of speed. Therefore we do not want client classes to be modified nor even recompiled when the developer switches forth - and even possibly back - between the symbolic and the concrete classes. Our objective is to reduce the impact of change through the use of *prototype objects*(3).

The notion of prototype object stems from *single-hierarchy languages* such as Self [Ungar&Smith Randall 87]. Single-hierarchy means that there is no distinction between a class and an instance. Moreover, classes are not viewed as static entities, but as real values: They are said to be *first class citizens*(4). In the case of OOMP, this helps selecting provider classes after compile-time. Thus it is no longer necessary to edit any source file when the developer wants to switch from one implementation to another. We may say that it indirectly reduces the risk of introducing typographical errors or other incoherences.

Our implementation language being class-based, we will have to simulate the mechanism of prototype objects: This is possible since every object has an inherent information which tells what class it belongs to. To exploit this, each abstract class will provide subprograms to set and test which of its derived classes has been selected as implementation.

The selection of an implementation may be performed either by the developer himself in the normal source text, or indirectly, by generation of a *configuration module*, which acts as a universal client in the client-supplier graph (see Figure 5), and which will be activated at program start-up.

3. This name clash is unfortunate, but corresponds to the normal usage. [Coplien 92] proposes the appellation *exemplar object*.

4. Smalltalk [Goldberg 84] is however an example of class-based language where classes are first-class citizens.

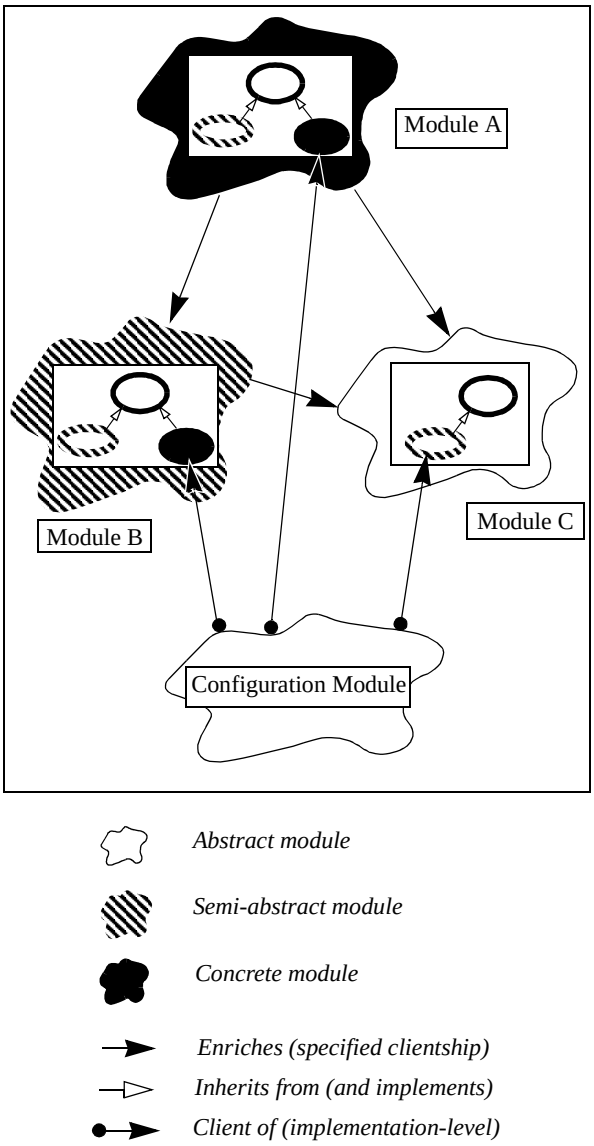


Figure 5. Role of the Configuration Module

Figure 5 shows a situation where the developer decides to try out his prototype, while `Module A` is completely concrete, `Module B` semi-abstract, and `Module C` entirely abstract, i.e. treated symbolically.

The prototype object is typically used when it comes to instantiating values, in the functions implementing the constructors of a given class. Another situation, which may easily appear in algebraic specifications, is when a method has no argument which attaches it to the currently defined class; a possible solution is then to resort to the prototype object as

dummy argument. Figure 6 shows the pattern which is used in OOMP. We could have explicitly shown a method for copying objects, but this is usually provided by programming languages under the form of a copy constructor. The class *SubClassName* may as well be the automatically generated symbolic class, in which case the methods $Operation_1$ to $Operation_o$ will not be redefined.

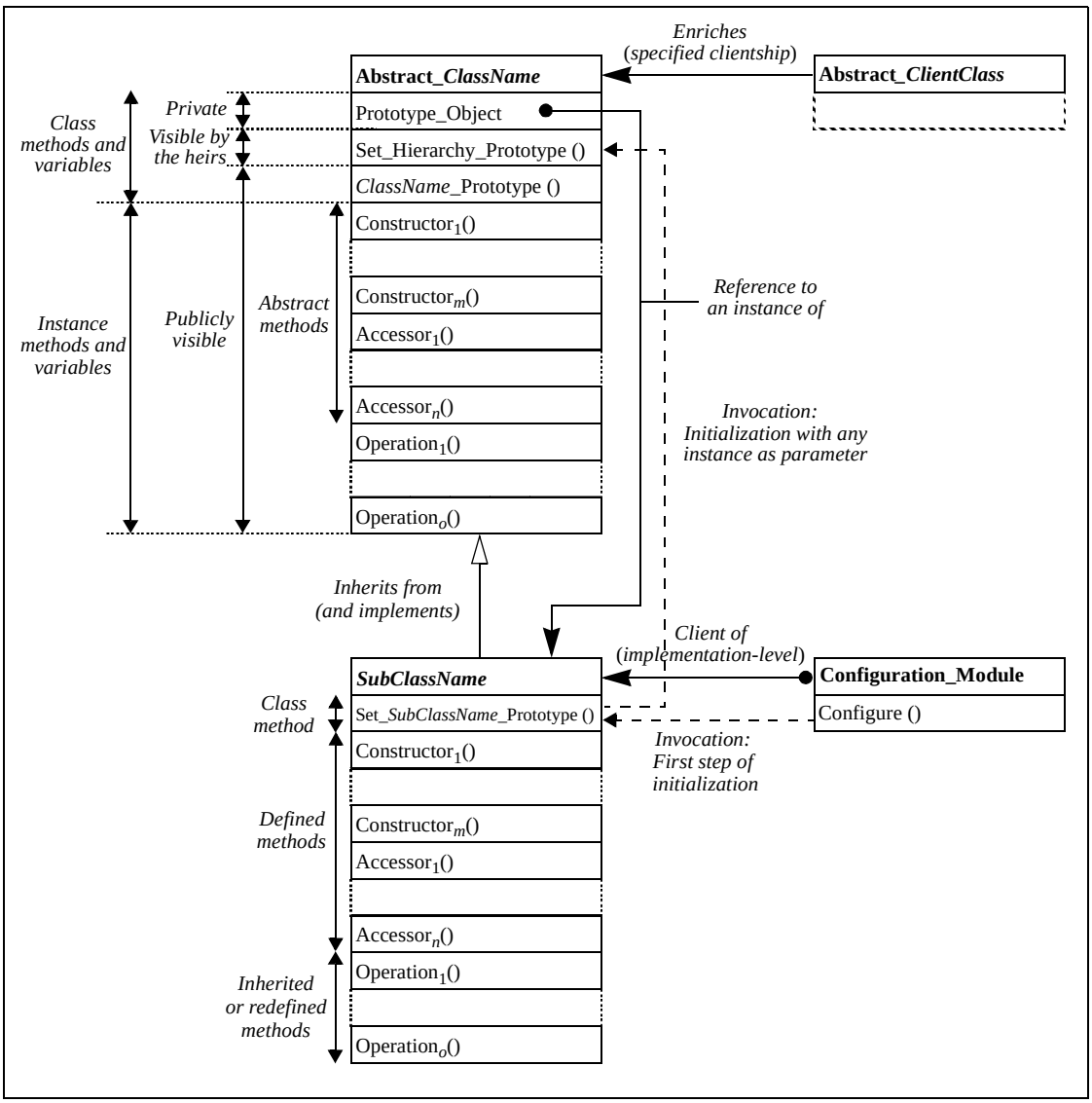


Figure 6. Detailed View of the OOMP Class Pattern

By decoupling the client from the provider we obtain the dynamics and the flexibility that are needed for a prototyping method to be attractive. This use of object-orientation is related to the general notion of *open implementation* [Kiczales 96], where the objective is to allow

the client to decide about some implementation details of the modules it uses. This is most easily realized through *reflexive languages* and *meta-object protocols* (see e.g. [Ichisugi, Matsuoka&Yonezawa 92]), but may be simulated in conventional languages like C++ [Stroustrup 91] and Ada95 through the use of *creational patterns* [Gamma et al 95] as we do[(5)]. The term *creational* comes from the fact that we essentially want to overcome an ultimate restriction to real scalability of class-based languages, namely the fact that one must know the name of a class in order to create instances of it. Our use of prototype objects allows the prototyping tool to generate abstract classes serving as protocol class, i.e. a unique interface to the outside world for all the possible concrete implementations.

2.7 The Incremental Prototyping Process

Now that the generic mechanisms for the support of OOMP have been presented, we can formulate a general development model. The approach taken in SANDS/CO-OPN proposes a formal and incremental method which covers:

- Modeling by the means of an object-based (or object-oriented in [Biberstein&Buchs 95]) specification language supporting concurrency, refinement, analysis and simulation [Buchs&Guelfi 91].
- Automatic code generation with a prototyping tool, as well as incremental incorporation of hand-written code portions in order to obtain a satisfying end-user implementation [Buchs&Hulaas 96].
- Specification-based test generation assorted with a test reduction technique founded on the introduction of hypotheses about the program [Barbey, Buchs&Péraire 97].

Figure 7 gives an overview of the activities and formalisms involved in the SANDS/CO-OPN methodology [Buchs 96].

5. Our pattern lies somewhere between the *prototype* and the *abstract factory* of [Gamma et al 95]. Our specificity is that the abstract class serves as prototype manager for its own hierarchy, and that the prototype is not only used for object instantiation, but may also automatically provide a dummy receiver argument (i.e. a target object) when needed, which results in public function profiles that are closer to the formal specification.

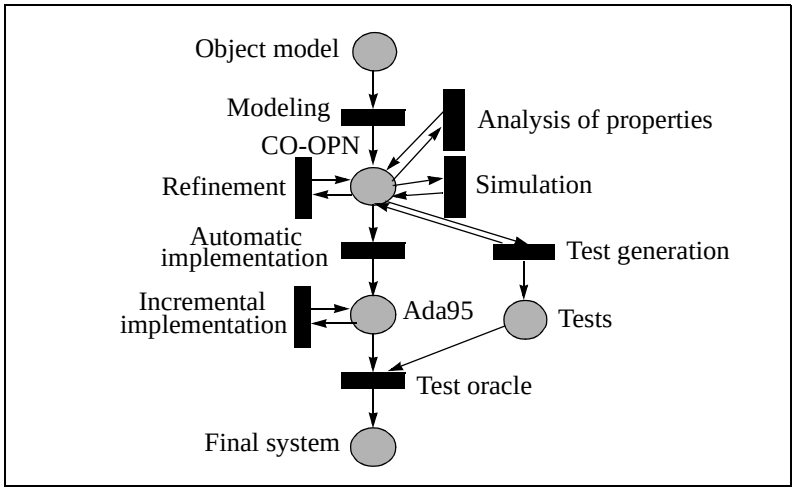


Figure 7. Activities and Formalisms within SANDS/CO-OPN

There is no precise method on how to obtain a CO-OPN specification from the user's requirements. In general, there has been little research on the conjoint use of the transformational model with the waterfall or the spiral model [Boehm 88] in order to guide the refinement of formal specifications, although it is recognized that both approaches are complementary [Sommerville 92]. One exception is the work presented in [Kemmerer 90], who distinguishes three levels of coupling:

1. The *after-the-fact* method, where the formal specification is produced at the end of the development, just for verification purposes. This approach is very expensive, since the formalization step does not profit to the other phases, and is therefore advisable only for safety-critical projects.
2. The *parallel* method, where both models are followed at the same time, is in principle executed by two dedicated teams, and requires a constant exchange of information between both processes. This approach allows speeding up the time to delivery of the final system, under the condition that the teams are well synchronized. Here, the benefits of the formal methods may be exploited during the whole development process.
3. The *integrated* method is presented as the most cost-effective. Its principle is to have a single process and to report all design decisions directly in the formal specification language.

The last approach is interesting, but probably a bit over-enthusiastic about the possibilities offered by formal notations. Our position is that it does not dispense from producing documents in a more conventional fashion in order e.g. to describe the non-functional aspects or to have a written trace of the justifications for the different decisions.

In [Sinclair, Clynch&Stone 95] a methodology is presented, which uses OMT [Rumbaugh et al 91] for analysis, “OMT*” for preliminary design, and SDL [CCITT 88] for detailed design. Their intermediary “OMT*” dialect has well-defined semantics by opposition to the original OMT notation, of which it is a subset, and helps filtering the constructs which have no equivalent in SDL. The translation into the formal SDL description is semi-automatic, allowing finally the validation of the system with a simulation tool. Automatic implementation is not provided.

A recent contribution, [André et al 96], resides in a structured and formal development method for implementing reactive real-time systems in object-oriented languages. The process starts with OMT, with some extensions in order to represent spontaneous events in the object model, which is then translated into a formal graphical “SyncCharts” representation. This latter notation may be directly mapped into the Esterel language [Berry&Gonthier 88], which is itself compiled into C++ by passing through two other intermediate translation steps. Although a certain continuity is ensured from OMT to C++, in the sense that there is no notational gap, it is not very clear in which phase of the development the different notations are used, neither at which levels prototyping is possible.

Currently ongoing research in the SANDS/CO-OPN group aims at integrating object-oriented development methods such as OMT or Fusion [Coleman et al 94] with CO-OPN. An additional difficulty in this context is to take into account the concurrent and distributed aspects in the modelled systems. Neither the original version of OMT nor that of Fusion address modeling of distributed systems. The second generation of OMT has however adopted some notations for concurrency [Rumbaugh 95]. It is therefore not our intention to enter into more details about this subject and provide precise answers to these problems. We will limit ourselves to indicating that, from a technical point of view, it is advisable to perform a detailed design phase prior to the automatic code generation step (see sub-section 2.7.1). This is also the approach taken in [Kordon 92].

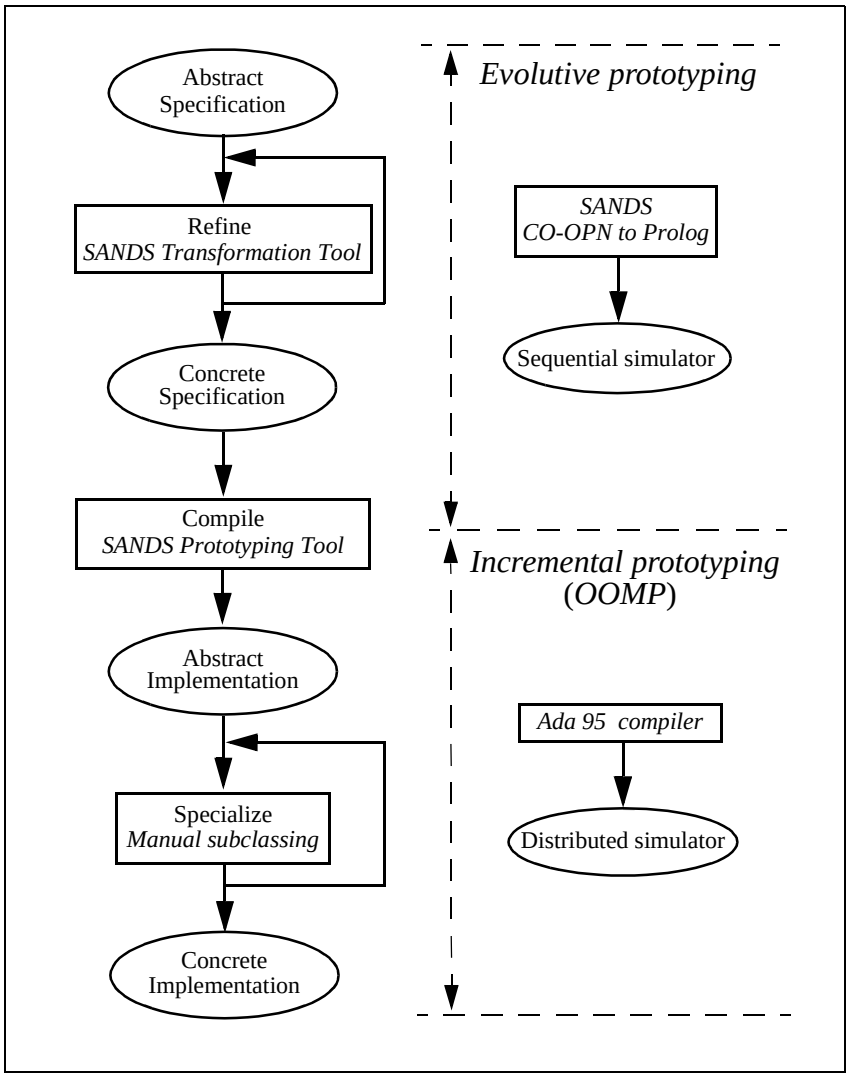


Figure 8. Our Proposal for an Incremental Prototyping Methodology

Figure 8 shows the development process we propose in relation with OOMP, by emphasizing the fact that the prototype may be executed at any moment: The specifications may be either simulated with an inference tool during the evolutive prototyping phase, or the code compiled and executed on a network of workstations during the lower-level incremental prototyping phase. The so-called *concrete specifications* are obtained partly by classical design techniques and partly by correctness-preserving refinement steps, e.g. with semi-automatic transformations which increase the level of parallelism of the resulting system [Buchs et al 96].

2.7.1 Why Object-Oriented Design is Necessary

As already mentioned, and as will be developed in chapter 5, there may be some technical problems when it comes to translating specification formalisms which are *not* object-oriented. The difficulty lies in the fact that, e.g. in algebraic specifications, the operations are not necessarily attached to a specific type, whereas this is compulsory to be mapped correctly into an object-oriented programming language. In those cases it will be indispensable to add a parameter, the *controlling argument*, to create an artificial link to the currently defined class. Another unclear situation is when an operation definitely belongs to a particular class, but the latter is defined in another module: Statically typed languages like C++ and Ada95 declare such cases illegal. The prototyping tool may then circumvent the problem by silently defining a new class to host the operation, or make the operation a global function, i.e. which depends on no specific class and which can therefore not be redefined by the developer in the concrete code.

To avoid these shaky constructions, one solution asserts itself: Before submitting a formal specification to the compiler, it must undergo a phase of *object-oriented design*, for instance with the Fusion method [Coleman et al 94], in order to make the interfaces compatible with the idioms of the object paradigm. In the case of CO-OPN, this rule applies only to the algebraic specification part since, concerning the Petri net part, a module is in itself a type definition and has therefore naturally adequate interfaces.

2.8 An Error Detection Scheme for the Concrete Code

A Basic Idea and Some Applications

In our version of mixed prototyping, the methods of the abstract classes, which are guaranteed correct w.r.t the specifications, are inherited by the concrete classes, and do no longer necessarily disappear when the developer wants to redefine them. This is because we exploit inheritance instead of genericity. Moreover, since the interfaces of the specifications are known, it is possible for the prototyping tool to derive new classes from the user's hand-written code. These classes may then be put to work transparently, because the consequence of dynamic binding is that the clients do not know which subprograms they are really calling. This opens the way to new techniques for tracing the execution and testing the correctness of the concrete code.

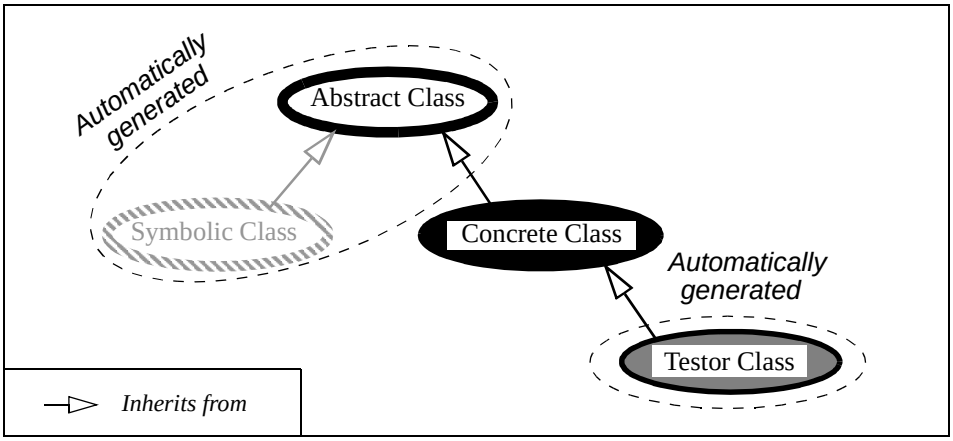


Figure 9. Position of the Testor Class in the Hierarchy

Figure 9 shows how the concrete class is “sandwiched” between two automatically generated classes. Some conceivable uses of this possibility are:

- To test that the operations of the abstract and the concrete classes have the same semantics by calling them successively. This requires some means of comparing the results of the operations, as well as the state of whole objects in formalisms which are not restricted to immutable values.
- To perform the same test as above, but in a way which is less costly: Instead of comparing states, the testor class can verify that some pre- or postconditions are respected. This solution is somewhat more difficult to implement automatically, but in compensation there should be no noticeable slowdown at run-time.
- To monitor all the calls to the concrete class. We could imagine special classes writing traces to files, or other tasks which are more conveniently fulfilled during normal execution than in a debugger, especially for distributed programs.

A Scheme for Automatic Generation of Executable Assertions

The test schemes of the two former points may simply be assimilated to the technique of *executable assertions* which is enforced in defensive programming styles. An assertion is a statement about a precondition, a postcondition, or an invariant, which must always hold, otherwise the program must take some exceptional measure, like for instance to halt or to transfer the control to an exception handler. This means that the program or module is run as a whole, exactly as in normal conditions, except that special checks are performed before and/or after each invocation to a method of the tested class.

Techniques based on executable assertions may be exploited during different phases of the software life-cycle [Rabéjac 95]:

- In the operational phase: [Andrews 79] proposes to use them for filtering input data of a radar real-time control software in order to protect the software from hardware errors.
- In the testing phase: [Mahmood et al 84] describe an experiment which consisted in testing a flight control software by the means of executable assertions, in order to detect design errors.

For this approach to be attractive, the cost of checking the assertions must be very low. This cost depends on how the assertions are built - which in turn depends on the nature of the specification language - and where in the control flow they are placed by the prototyping tool. We will give more concrete ideas about this approach in chapter 5 and 7, which are devoted to the application of OOMP within specific formalisms.

Although we have not had the time to further explore the issue, we have good reasons to think that the class pattern described for generating executable assertions may also, with a few changes, provide an integrated framework for performing dynamic specification-based testing: We have a test execution environment in the abstract code, which dispenses from writing stubs and driver modules, and we have an oracle implemented by automatically generated comparison operators.

The Problem of Correlated Errors

The scheme is however not perfect due to the fact that the comparison of states or results (e.g. by an abstract equality operator) relies on the usage of hand-written constructor and accessor functions. There may be wicked situations where an erroneous accessor compensates for a bug in a tested operation, and thus “corrects” it. This is called a *correlated error* and cannot be prevented. On the other hand, accessor functions are usually rather simple functions, and therefore less error-prone.

2.9 Rationalization of Memory Management

From the the programmer’s point of view, object-oriented languages are also interesting because they help relieving client modules from the burden of destroying objects created dynamically. Some environments offer a general-purpose garbage collector, as in Eiffel [Meyer 87], which is precious in the frame of prototyping, since it helps concentrating on the problem to solve instead of on secondary implementation details. Unfortunately this induces a hidden cost, which may reveal unsafe in critical applications. In other languages, the root class can be used to implement memory reclamation, or can inherit this capability

from other classes. Ada95 provides two predefined root classes, the *controlled* and the *limited controlled* classes, whereas C++ allows the programmer to redefine the *new* and the assignment operators. In the latter two cases, one basically has two options:

1. Let the root class take care of the integrality of memory management;
2. Just use the root class to define a global frame, and implement the details of memory-management inside the derived classes.

In the case of mixed prototyping, the second solution seems to be better adapted since the internal representation can change radically between a symbolic implementation and its corresponding concrete versions. For instance, for AADTs we use linked lists as supporting representation during their symbolic treatment. They will however often receive a more compact form, without pointers, in their corresponding concrete instances.

It is beyond the scope of this work to treat memory management in more detail, as this subject depends on the source formalism, the target language, and also on the specific application being developed.

2.10 Assessment of OOMP

We think that our incremental prototyping technique is realistic in the sense that it does not only work on toy formalisms. We show in this thesis that it can be applied to the integrality of CO-OPN, which is a very rich specification language including extensive structuring possibilities, concurrency and non-determinism. Let us now try to discuss some potential charges against our approach.

2.10.1 Conditions for Semantic Validity

For the mixed prototyping approach to be formally valid with respect to the semantics of a given source language, then the latter must of course permit some form of refinement and not be too restrictive in its acceptation of semantics preservation. The chosen formalism should allow all the intermediate stages between the abstract and the concrete implementations, in other words, replacing just a fraction of its “normal refinement unit”. For instance, partial implementation of a signature, for algebraic specifications, or partial implementation of an Object module for the Petri net part of CO-OPN.

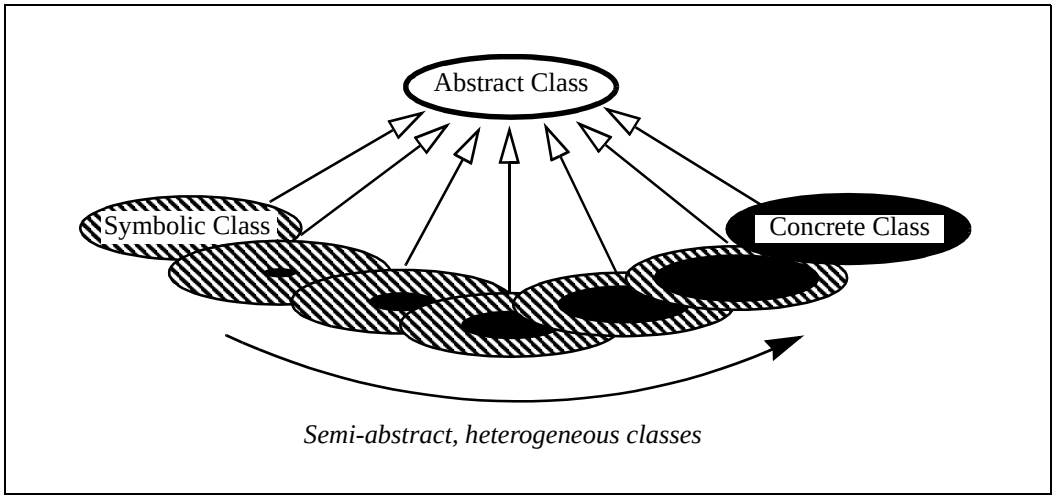


Figure 10. The Continuous Evolution from Symbolic+Abstract to Concrete

In the case of algebraic specifications, the paper [Choppy&Kaplan 90] stated the correctness criterion as being the following. Let be two different algebraic values of a partial model(6) defined only by the sorts and operations with concrete implementations. Then they must keep the same equivalence relation in the models obtained by incorporating the abstract definitions of the remaining operations. This criterion concentrates well on the heterogeneous aspect of the evaluation, i.e. the fact that potentially only a subset of the operations of a sort are concrete. This contrasts with all other existing approaches where the implementation granularity is the module and not the operation.

Whereas the above-mentioned proof of implementation correctness w.r.t. the algebraic specification is given on the model level, i.e. on state based homomorphisms, there exists another viewpoint where equivalence is based on observable behaviours (as in [Hennicker&Schmitz 96]).

In the case of CO-OPN Object modules, refinement is based on the relation of *strong concurrent bisimulation* (see section 3.7). This property is however defined on the basis of *markings*, a notion of state which is not modular. This unstructured approach implies that fractional replacements of Objects are directly taken into account by the semantics, as long as the algebraic model remains *unique* (definition 24 on page 83). For us, this uniqueness is guaranteed by the fact that the generated implementation of the algebraic part is based on term rewriting systems and by the condition that all hand-written variants stay within the same model, hence the model is always the *initial model* (see [Ehrig&Mahr 85] for a definition): This is perfectly standard in implementations of algebraic specifications.

6. Informally, a *model* is a correct implementation of an algebraic specification. A formal definition will be given in 3.6.1.

On the programming language level, strong and static typing contribute to the correctness of the concrete code. It is nevertheless the duty of the programmer to ensure that the semantics are truly respected. For instance, in the concrete code of a CO-OPN Object, one might increase parallelism by firing concurrently events which are specified as sequential but are in fact independent. A given Object must however keep strictly identical criteria for deciding about his willingness to accept incoming events (when evaluating the guards): The observable behaviour of offered services must not change. In other words, it is allowed to ameliorate the response time of a system as long as the resulting state in the concrete Objects is equivalent to what it would have been in the corresponding abstract Objects.

2.10.2 Implementation of Object-Oriented Formalisms

The formalism we demonstrate OOMP on is in fact object-based. Genuine object-oriented specification languages still constitute an active research field - see for instance CO-OPN/2 [Biberstein&Buchs 95] - and it is hard to tell what kind of mechanisms they will finally offer. The question is whether OOMP will interfere with the specification-level inheritance hierarchies. Take for instance a `Stack` specification, having a derived `Bounded-Stack` class specification.

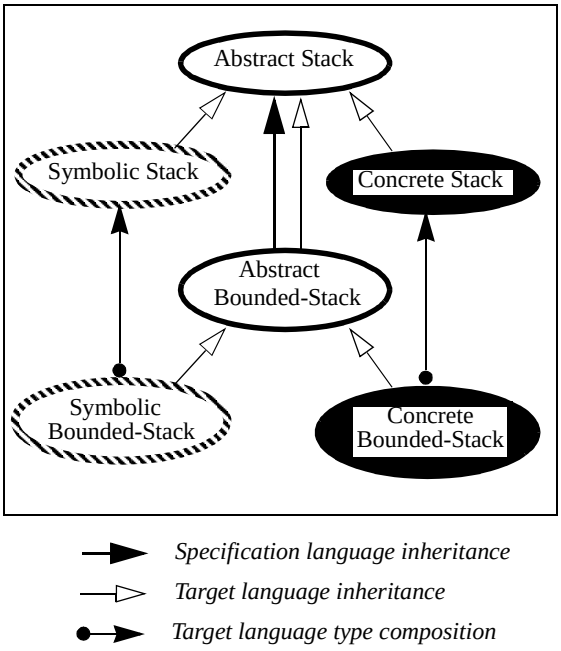


Figure 11. A Possible Implementation of Specified Inheritance

Suppose that `Concrete_Stack` was implemented as a bounded array. Then, if this is compatible with the specifications of `Abstract_Bounded-Stack`, it would be natural to

reuse this code for the `Concrete_Bounded-Stack`. We do not need multiple inheritance to do this, but as indicated in figure 11, only type composition, inducing a simple client-furnisher relationship between those two concrete classes. Therefore OOMP will at least not introduce repeated inheritance schemes which did not originally appear inside the source specification hierarchy.

Another problem would be to conveniently and automatically map specification languages with multiple inheritance such as VDM++ [Dürr&Plat 95], into single inheritance programming languages(7). The object-oriented version of Ada [Ada 95], which the design team for several reasons chose not to provide with multiple inheritance, has instead a set of specific constructions, relying essentially on genericity, to cover some idiomatic forms of multiple inheritance. The challenge would then be to identify these cases - probably not without some hints from the developer - and to translate them accordingly.

We do not further develop this discussion, because it will invariably lead to both source and target language-specific considerations. It is quite possible that object-oriented specification idioms will evolve to much more powerful mechanisms than what can be directly mapped onto current programming languages.

2.10.3 The Inheritance Anomaly in Target Languages

Most concurrent object-oriented languages present the problem that code reuse is not possible when redefining inherited procedures which contain code for concurrency control. This is called the *inheritance anomaly* [Matsuoka&Yonezawa 93]. The only solution for users of these languages is to copy textually the inherited code and then to add the code which is specific to the sub-class. Some research languages have solved this problem by separating the functionality from the concurrency control, to which a declarative form must additionnally be given [Frølund 92].

Since our intention is to support OOMP for distributed systems, it means that this problem will be met as long as conventional programming languages (such as Ada) are used for the implementation of prototypes. As a matter of fact, opportunities for code reuse are rare in the implementation scheme we propose for CO-OPN Objects (see subsection 6.5.2.3).

2.10.4 Possible Sources of Inefficiency at Run-Time

We distinguish mainly two potential sources of inefficiency during the execution of prototypes: the extensive usage of dynamic binding and the support for non-determinism.

7. There exists a tool [Plat&Voss 95] which compiles VDM++ into C++ (which itself provides multiple inheritance).

Slow-Down due to Object-Orientation

It is clear that in the current state-of-the-art in compiler technique, object-oriented programming incurs a certain overhead. This is mainly due to the finer grained encapsulation units and the use of dynamic binding. There does however already exist possible remedies: Some Eiffel compilers are able to transform dynamic into static binding [Meyer 87]. Work on eliminating unreachable procedures in object-oriented programs, such as [Srivastava 92], focus on lowering the size of the resulting binary code. The BRUNEL project [Simons, Kwang&Mei 94] aims at several optimizations such as link-time detection of class hierarchies where only one derived class provides instances. This allows collapsing the inheritance hierarchy and performing static resolution of procedure calls, reducing simultaneously both the code size and the alleged slowness. The *flat* tool of Eiffel has the same effect by operating on the source code level.

The Cost of Non-Determinism

A useful specification device is the implicit concept of *search non-determinism*[8]. It allows the developer to concentrate more on the *what* than on the *how* because it eliminates the need for all the control structures of procedural reasoning.

This comfort has unfortunately a certain cost at run-time. Search non-determinism is typically implemented by *backtracking*, a mechanism which is uneasy to reproduce in imperative languages. The difficulty comes mainly from the requirement that the search space must be explorable exhaustively - this is especially hard in distributed systems - and preferrably without programmer intervention.

The way we propose to implement backtracking (in chapter 6) is essentially based on safety and extensibility principles. It is possible that other solutions, attained by further *reification*, i.e. conversion of any entity to the class construct, may achieve better performance, but at the price of making the mapping between the specification and the generated code less intuitive. We think however that most of the inefficency will disappear at completion of the mixed prototyping process, because we give the programmer the tools for manually transforming backtracking into procedural control structures. The performance of the resulting system will then finally depend essentially on the nature of the application, i.e. to which extent it is search-oriented.

A currently successful approach in the field of logic programming is to annotate the definitions with determinism indications (see e.g. *Mercury* [Henderson, Somogyi&Conway 96]). Thus, it is always possible to know whether several solutions must be expected from a predicate evaluation (or method call) and to optimize by preparing the run-time environment

8. Also called *don't know non-determinism*.

accordingly. This alternative could reveal itself particularly fruitful in a distributed environment, where the delays due to remote procedure calls are especially perceptible.

In a more general sense, it is possible that the structuration patterns imposed by the OOMP scheme may hinder some optimization techniques developed for the implementation of certain specification languages. In the work we have done on algebraic specifications and on Petri nets, the only meaningful problem seems to be the above-mentioned non-determinism.

2.11 Putting OOMP to Work

In this section we will briefly give some more concrete indications about the interconnection of the tools which are to support our form of incremental prototyping within the SANDS environment [Buchs et al 95], as well as the additional mechanisms by which the developer gives implementation hints to the specification compiler.

2.11.1 General View of the Prototyping Tool

The prototyping tool is made of four parts, as depicted in figure 12 below:

- The CO-OPN checker[(9)], which executes lexical, syntactical and semantic analysis of the source specifications written in CO-OPN. This module is common to all the tools, which must thus be able to use the generated syntax tree;
- The front-end of the prototype compiler, which carries out additional semantic tests to verify that the specification is concrete enough for the incremental prototyping phase. It also records all user annotations written in separate files;
- The back-end(s) of the prototype compiler, which performs the proper compilation of the specification by translating them into a procedural form. We may say that there is one main compiler, and a set of auxiliary smaller modules, the role of which is e.g. to prepare the testor classes;
- The code generator(s), which produces the class hierarchies in the target language. There must be one code generator for each target language. Essentially their work consists in adequately “packaging” the code into class and function definitions. An important aspect is that they must have a perfect knowledge of the lexical conventions of their target language in order to transcribe all identifiers of the source specifications into compilable and easily recognizable identifiers in the generated code. This latter role is important for the prototyping to proceed smoothly.

9. This part was implemented by Mathieu Buffo.

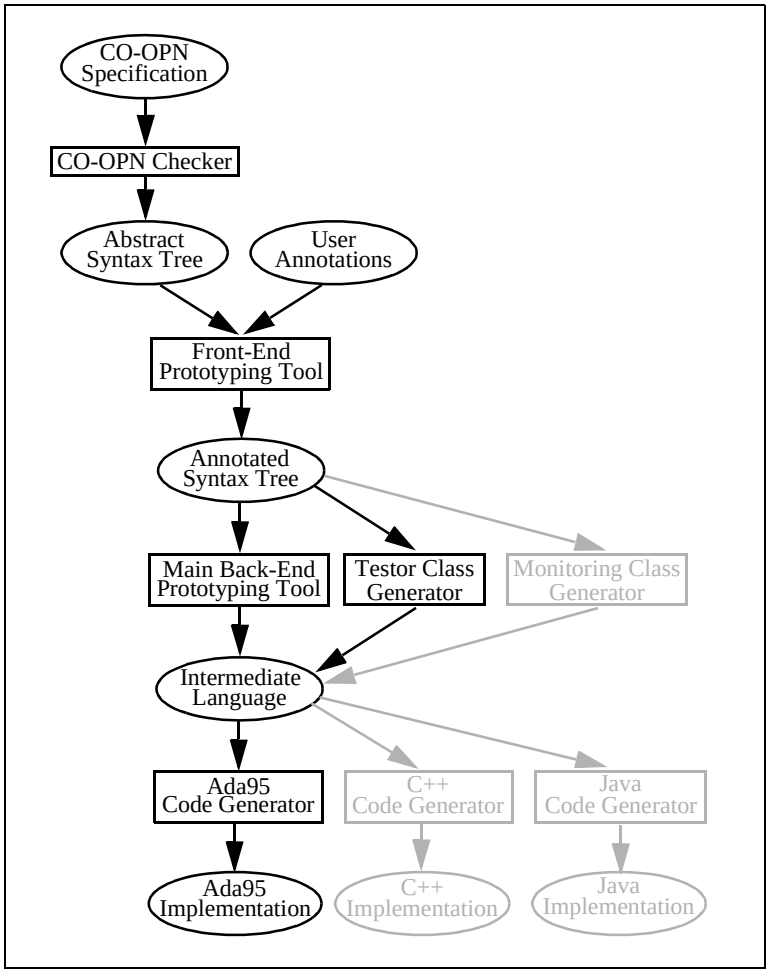


Figure 12. Operations and Outputs in the Prototyping Tool

2.11.2 The Annotation File

In the previous sections, there has been several allusions to the possibility of easily switching between implementations. But nowhere did we see how this was actually done: That is the purpose of this sub-section.

Remember that the source language is CO-OPN, in which the dynamic properties of a model is expressed by the means of modular algebraic Petri nets. Each Petri net Object defines a sub-net which is connected to other sub-nets through synchronization links. Algebraic abstract data values can be transferred during the synchronization.

In this thesis it has been decided that the Petri net modules are to be implemented as separate (coarse grained) processes executing on different nodes. Each process may use a locally defined implementation of the data types it processes, which means that there is a large support for heterogeneity between processes: This ability is described in section 5.6.1.

```
ANNOTATION Network;
INTERFACE
   USE
      Message -> TEST "Packed_Message",
      ID -> "Abstract_ID";
   METHODS
      Put _ _ _ : message id id -> IN IN IN;
      Get _ _ _ : message id id -> OUT IN OUT;
BODY
   ;; No private declarations to annotate
END Network;
```

Figure 13. Annotation File for Object `Network`

Figure 13 gives an intuition of how an annotation is associated to an Object called `Network`, the source of which is given in Figure 19 on page 57. Briefly, we can see that the arrows indicate different kinds of mappings:

- In the **USE** part, the arrow maps the name of an imported module to the corresponding implementation. The keyword **TEST** tells the prototyping tool to generate testor classes derived from the indicated implementations.
- In the **METHODS** part, the arrow associates with each method a strongly moded profile. This is necessary as logical variables are not supported in the prototyping process, for safety and efficiency reasons, and also because it yields code which is easier to interface with (more on this in section 6.5.2.2).

Depending on the situation, more indications may be necessary, for instance if we want to reuse precompiled or non-modifiable libraries which do not follow the naming conventions of our prototyping tool. In any case it is a good idea to keep the annotations separate from the specifications themselves in order to guarantee a maximal independence between the different development tools: We could for instance have other kinds of auxiliary files for associating test sets with each module. This approach also allows us to have several configurations for a given implementation.

2.12 Open Problems and Possible Extensions

These are some closely related problems that we have not had the time to examine in this thesis:

- The possibility of integrating existing software components, which are not formalized, by e.g. introducing new enrichment declarations in CO-OPN. At the strict level of the specification language this must be completed by writing interfaces for the external components. From a programming language point of view the situation is more delicate since the calling conventions and the internal representations of common data types must be taken into account, especially in mixed language environments. This subject has already been considered on the formal level in [Kordon 92], and on the programming language level in [IPTES 94] and [Fröhlich&Larsen 96].
- In a related area, mixed prototyping may be used to help reengineering of existing systems, to which it is desirable to retro-fit complete specifications (and not only interfaces). The idea is then to remove errors, either in the specifications, or in the existing components, by detecting different behaviours between the abstract code and the concrete code, which is implemented by the original modules [Cherki&Choppy 96]. To put this concept into practice, the above-mentioned problem of heterogeneity in the calling conventions and data structures must first be resolved. It also needs some convenient way to compare the behaviours: This may be done by the dynamic code verification principles presented in this thesis. The paper [Zaremski&Wing 95] defines a formal specification-based classification for determining how well two software components match. They use pre- and post-conditions for establishing a scale ranging from syntactic interface compatibility to true substitutability.
- It is necessary to map the specifications into equivalent programming language constructs in an intuitive way, a requirement which is not present in other forms of prototyping. This subsumes that easily recognizable identifiers may be generated. Ada is fortunately a generous language which allows many kinds of overloading, therefore identifier *mangling* (coding in order to ensure uniqueness) is generally not needed. Mixed prototyping may be less comfortable in languages, such as Eiffel, which do not support overloading.

2.13 Related Work

Since this is an introductory chapter on OOMP, we focus on related work in the general fields of incremental prototyping and executable assertions. The chapters devoted to more technical aspects of implementing algebraic specifications and CO-OPN Objects will contain their own respective summary of related work.

It has long been recognized that object-oriented languages are suitable for prototyping in all generality. Smalltalk [Goldberg 84] is usually the favourite in this context because of the flexibility offered by its dynamic typing and its wealthy environment (see e.g. [Krief 92]). Class derivation and specialization are the keywords for the support of incremental development, the root classes being considered as the skeleton of the application. Another interesting aspect of Smalltalk is the concept of *meta-class*, which lets the programmer

transparently modify the predefined mechanisms of the language, in a manner akin to reflexive languages. This property allows quite elegant formulations of service layerings such as fault-tolerant protocol stacks.

Nevertheless, because of its relative inefficiency, Smalltalk is more often used for throw-away prototyping, the real application being then developed from scratch. Further, none of these approaches are based on formal methods, nor on automatic generation of incremental and heterogeneous prototypes.

2.13.1 Incremental and Heterogeneous Prototyping

Within this domain of prototyping, we can distinguish the works which additionally allow concretizing the generated code from those that don't.

In the formal methods community few approaches really try to tackle the problem of automatically implementing end-user applications. Instead, they usually only support simulation, in the sense of specification animation. There is also some confusion due simply to the use of ill-defined terminology. In the case of Petri nets, one can often hear correct but misleading sentences like: "The implementation is a simulation of the token game".

For this reason, let us establish the following definitions: We will define *simulation* as the process of animating a specification in order to observe and analyze its behaviour. This is usually done in logic or functional languages within a closed environment (the development tool), but also sometimes as stand-alone programs on dedicated parallel architectures. *Implementation* aims at direct generation of application programs described by formal specifications. For efficiency reasons, prefence goes rather to the use of compiled procedural languages.

Prototyping without Code Concretization

We consider here successively methods based on Petri nets, on algebraic specifications, and finally on other formalisms. Interestingly, the two former notations are the most prolific domains, because they contain comparatively more operational information.

- Some Petri Net Approaches
  On the subject of prototyping based on Petri nets, we can mention e.g. [Bréant&Pradat-Peyre 94] which generate implementations in *OCCAM* [INMOS 88] for parallel Transputer machines. It is therefore questionable whether they do not address simulation more than implementation. An example of research project which is more software engineering-oriented is [Kordon 92], which uses Ada83 as target language. Interestingly, in later papers, the Petri net formalism is abandoned in favour of a more expressive semi-formal language, and Well-Formed Coloured Petri nets [Chiola et al 91] are used as an internal representation on which analysis techniques may in turn be applied

[Kordon 94][Kordon 95]. An approach where object-orientation is exploited in the target language is [Holvoet&Verbaeten 95], where the inheritance mechanism is presented as a means for the developer to manually extend the basic class of models supported by the source formalism and/or the prototyping tool. For instance, *inhibitor arcs*[10] may be built by a derivation of the predefined *arc* class. This would also be possible to some extent in OOMP, although we consider that the semantics should be strictly preserved in the concrete code.

- Algebraic Specifications

  In the domain of algebraic specifications, there are some tools, e.g. [Garavel&Turlier 93], where the abstract code, usually written in C [Kernighan&Ritchie 78], may be generated so as to call hand-written functions, but the level of interaction is fairly limited, not to talk about the resulting type safety. The work which is described in [van der Meulen 90], although they use the term *incremental implementation*, is not aimed at implementing algebraic specifications, but rather at their incremental rewriting and interpretation in interactive environments.

- Other Formalisms

  We can mention for instance the Statemate tool [Harel 90], which is capable of generating prototypes in Ada or C on the basis of Statechart graphical specifications [Harel 87]. These prototypes may be combined with external modules *at link time* only. The VDM++ Toolbox [Plat&Voss 95], which generates C++ classes from VDM++ specifications, requires the integration of hand-written C++ code for the parts of the source formalism which are not executable, such as implicitly defined methods. In [Mañas&de Miguel 88] a LOTOS [ISO 88] to C compiler is described where the generated code may call hand-written routines and vice versa. The Ptolemy tool [Buck et al 94] is a generic framework for system-level design which allows mixing models of computation. It is made of an extensible set of *domains*, of which each domain contains the description of a model of computation and its associated communication and scheduling rules. Therefore it supports the simulation of heterogeneous prototypes without being itself restricted to a particular formalism. A notation which is quite popular, Z [Spivey 89], is not appropriate for automatic implementation, because it does not provide enough operational information. Only Prolog-based simulators have been built for Z.

Prototyping with Code Concretization

To be considered within this category, tools or methods must additionally be designed for incremental replacement of abstract modules by concrete modules. In fact the only candi-

10. *Inhibitor arcs* allow the firing of a transition only if its input place contains a number of tokens inferior to a given value. This is the opposite of the behaviour of a standard arc, but provides more expressivity.

dates, mixed prototyping excluded, are the IPTES project [IPTES 94] and the directly related [Fröhlich&Larsen 96].

The European IPTES (*Incremental Prototyping of Technology for Embedded Systems*) project, which started in 1990 and lasted three years, had as objective to apply Boehm's spiral process model [Boehm 88] in the field of real-time systems. The result was a methodology and a supporting environment [Elmstrøm 93] for assisting in the development of embedded real-time software as series of prototypes being incrementally refined, from high-level logical description to code in the implementation language. IPTES prototypes are described in a specification language which is an extension of SA/RT diagrams [Ward&Mellor 85], complemented with descriptions in "Meta-IV", a language based on VDM [Jones 90], for specifying data and transformations. The timing requirements are expressed using a net formalism they call "High Level Timed Petri Nets" into which the SA/RT models are translated.

In the IPTES project, modules at different levels of abstraction may synchronize and communicate through the technology they call *run-time adaptation kernel*. Here resides the main difference with mixed prototyping: This approach requires converting the data structures at run-time for communication to take place between abstract and concrete environments. In fact, from a purely technical point of view, they really deserve the term of *heterogeneous*, while it is less justified for mixed prototyping. There always remains a clear distinction between the part which is simulated and the part which is implemented, while in our approach the frontier is less visible.

The *run-time adaptation kernel* has the advantage that mixed language and mixed paradigm interactions may be considered, resulting in potentially more important differences of abstraction between modules during the development. On the other hand, it is unlikely, in these conditions, that the concrete code may call functions of the simulated part. Only purely procedural and sequential concrete code is effectively allowed, resulting in a strict bottom-up development strategy. Last, but not least, there is no automatic code generation in the IPTES toolbox: The abstract part is in fact the simulator itself.

In subsequent work, presented in [Fröhlich&Larsen 96], the interaction between abstract and concrete code was ameliorated. Whereas the communication mechanism was previously based on C code, heavy-weight Unix processes and pipes, it was mutated to the combination of C++ code, single-process execution and dynamic linking for the integration of new implementations with the abstract parts. Instead of exploiting data abstraction principles, they still oblige the developer to study their internal representations in order to write the appropriate conversion routines. The specifications may now be automatically translated, and the concrete code is essentially seen as a means of providing the facilities which cannot be specified with their formalism, e.g. I/O and trigonometric functions.

In the above-mentioned related works, no one exploits the object paradigm for its natural capacity of having several implementations for a single abstraction. Moreover, there is nowhere a real prototyping phase on the concrete code.

2.13.2 Related Work in Executable Assertions

The principle of expressing assertions as predicates on abstract data types was described as early as 1972 [Hoare 72]. To put this idea into practice, the key is to have a mapping between the abstract values and the implementation values. This mapping is realized, in our approach, by the abstract constructor and accessor functions (depicted in figure 6) which delegate the low-level actions to the concrete class during the evalutation of an assertion. In the specific case of algebraic specifications, the automatically generated syntactic equality operators are built upon abstract accessors.

We consider from now on only tools and methods which are able to generate executable assertions on the basis of formal specifications.

A famous representative of this category is the *Larch project* [Guttag, Horning&Wing 85], which provides a formal description language to check that hand-written code does not violate the specifications. To this purpose, two complementary formalisms are used: The first one, the *Larch Shared Language*, is essentially the usual kind of algebraic specifications, and deals with the relationships between abstract data types and their associated operations. The second formalism is the model oriented family of *Larch Interface Languages*, which are extensions of conventional programming languages like Modula-2 or C++. An interface language provides the means of expressing invariants, input-output assertions about the functions being implemented, and functions to map concrete values onto abstract ones.

Anna [Luckham 87] is a formalism for annotating Ada sources in the form of comments. After preprocessing and compiling, this results in a program with executable assertion code which may raise exceptions. The adequate information for generating assertions is made directly available at the right place by the programmer. The role of the preprocessor is therefore trivial.

In the VDM formal notation [Jones 90], axioms may be stated explicitly and are easily identifiable by any specification compiler. They are however not used in procedural implementations, since VDM also provides a constructive part, which gives the operational information required for more efficiently executable programs. Among the mentioned axioms are pre- and postconditions: The VDM++ to C++ compiler [Plat&Voss 95] implements these as separate functions. If the developer wants to use them as assertions, he must edit the generated code and insert the calls by hand.

2.14 Epilogue

We have now seen the fundamentals of mixed prototyping and the contributions brought to it by a systematic usage of object-oriented principles such as data abstraction, incremental programming, dynamic binding and creational patterns. The presentation was intentionally very abstract, in order to show the generality of the approach. Some indications were given concerning the usage of OOMP as a general development methodology, while it is out of the scope of this work to elaborate the precise criteria for integrating a formal approach with more classical structured methods. We can now formulate a wish list for the application of OOMP to specific formalisms:

- It should be *intuitive*: The generated prototype must be structured adequately so that the classes and methods at the programming language level correspond to easily identifiable entities of the source formalism and the modelled system.
- It must lead to *correct implementations*: Since the generated code will permanently serve as reference, it must give the best possible guarantee. The use of assertions during the incremental concretization may also contribute to the quality of the resulting piece of software.
- The generated code should be *efficient* although it is possible to hand-tune it: The developer should not be obliged to redefine everything in his concrete code.
- The run-time support should be *robust* since we are not doing throw-away prototyping. It should provide the services required for the reliable execution of the resulting application, especially in distributed environments.

The remaining chapters of this report are covering the answers to these requests. In particular, we demonstrate how mixed prototyping with object-orientation is put to work with algebraic specifications and CO-OPN Petri net Objects.

Chapter 3

The CO-OPN Specification Language

3.1 Introduction

In this chapter we will perform a thoroughgoing presentation of the CO-OPN specification language. This is necessary to understand the specificity of its semantics, and in particular its original notion of object activity and synchronization. For this description we base ourselves on the original CO-OPN paper [Buchs&Guelfi 91] and also partly on the PhD thesis [Biberstein 97] from which it was possible to obtain a more polished formulation of the former semantics. We completed this work with numerous examples and remarks in order to orient the discourse in the direction of our more pragmatic approach of CO-OPN, namely automatic implementation and prototyping of distributed systems.

The concurrent part of CO-OPN is currently defined by the means of an abstract syntax and a very high-level form of operational semantics called *structured operational semantics* (*SOS*) [Plotkin 77]. Concerning the syntax we will from time to time refer to its concrete version too if we feel it may bring interesting insights into the more practical aspects of the language. More importantly, we will also make sporadic digressions from the given operational semantics to equivalent lower-level point of views because we think that this will provide a smoother transition to the algorithmic descriptions of chapter 6.

The SOS frame is directed towards deductive reasoning schemes built upon a set of predefined axioms, to which one adds the inference rules specific to the formalism under consideration. The resulting system may be used as a basis for calculating the allowed behaviours of a given specification, either by hand, or by resorting to dedicated tools such as the CO-OPN simulator [Buchs, Flumet&Racloz 93]. A prerequisite for such tools is therefore the ability to manipulate inference rules: This is most easily realized by implementing them in a logic programming language such as Prolog [Colmerauer 83]. The advantage of this solution is that a resolution procedure is dispensed directly by the language. In other words, the operational device by which correct reasonings are elaborated and appropriate variable assignments are obtained is supplied for free. The purpose of this

work is however to produce prototypes implemented in imperative programming languages, which means that we must elicit the resolution mechanism, since it is no longer predefined. It is not only this difficulty, but also the additional problems raised by the execution conditions met in distributed systems - lack of global state, hardware failures - which must be answered by the algorithmic design of chapter 6, hence the need for a progressive descent in the levels of abstraction.

The rest of the chapter is organized as follows. After the preliminary remarks on the evolution of CO-OPN, we provide a gentle introduction to the language through a small example. Then we gradually and formally introduce the different components of the formalism, first syntactically and then semantically, from algebraic specifications to synchronized CO-OPN objects. We assume however that the reader has some familiarity with classical Petri nets [Reisig 85]. Finally, a discussion about the nature of the language and the compositionality of its semantics will give the reader a better understanding of CO-OPN and help us proving the correctness of the implementation scheme given in chapter 6. It should be noted that in this thesis we emphasize the concurrent part of CO-OPN because the other facet of the language, multi-sorted algebraic data types, is now a well-established formalism. Therefore the corresponding compilation algorithm (see chapter 4) presents only minor changes compared to existing results, modifications which are needed to enable incremental prototyping.

Sometimes we will use the term *AADT* as a general abbreviation for *algebraic abstract data type*, while the name *Adt* will refer to specific syntactic or semantic entities of the CO-OPN language such as *Adt module*. Similarly, *object* is a generic denomination for an encapsulated Petri net, while *Object* denotes distinctively the module construct of CO-OPN.

3.2 Historical Background

The CO-OPN (Concurrent Object-Oriented Petri Nets) project [Buchs&Guelfi 91] was initiated in order to propose a formal specification language for the design of large concurrent systems. This goal is attained through the definition of a modular specification language combined with Petri nets. More precisely, CO-OPN objects are interconnected modular entities described by algebraic nets. A set of tools, the SANDS environment, has been built [Buchs, Flumet&Racloz 93][Buchs et al 95] to support this formalism, and comprises utilities such as a graphical editor, a transformation tool and a sequential simulator for assisting in the development of specifications.

The CO-OPN language is based on two simple and elegant concepts, *algebraic abstract data types* (AADTs) [Ehrig&Mahr 85] and *Petri nets* [Petri 62][Reisig 85], mustered into a more general model, *algebraic net*s [Reisig 91]. Although algebraic nets constitute an

important improvement over classical Petri nets, they are quite useless when facing large problems. Therefore, the following structuring facilities have been included in the CO-OPN model:

- modularity and encapsulation
- genericity
- synchronization and communication between modules

Before pursuing our presentation of CO-OPN, it should be made clear that several versions of the formalism have been elaborated during the research effort which started in 1989, and that they have different views of the notion of object-orientation. The first version, described in [Buchs&Guelfi 91] and [Buchs, Flumet&Racloz 93], is called CO-OPN v1.0 and does not provide inheritance nor dynamic object instantiation. It is therefore rather *object-based* in the classical sense (see [Wegner 87]). The next formulation of CO-OPN is version 1.5 and consists mainly in syntactical updates [Buchs et al 95]. Finally, CO-OPN/2 is the fully object-oriented manifestation of the language [Biberstein&Buchs 95]. Its definitive semantics have been settled very recently [Biberstein 97] and this is why this report is centered instead around the 1.5 version of CO-OPN.

3.3 Introductory Example: The Collaborative Diary

The formal model is based on an approach in which the notion of encapsulation is ensured by *methods* and abstraction is supported by *synchronization expressions* between the methods. An *object* is an algebraic Petri net, with *parameterized transitions* as methods and the *markings* of the places as internal states.

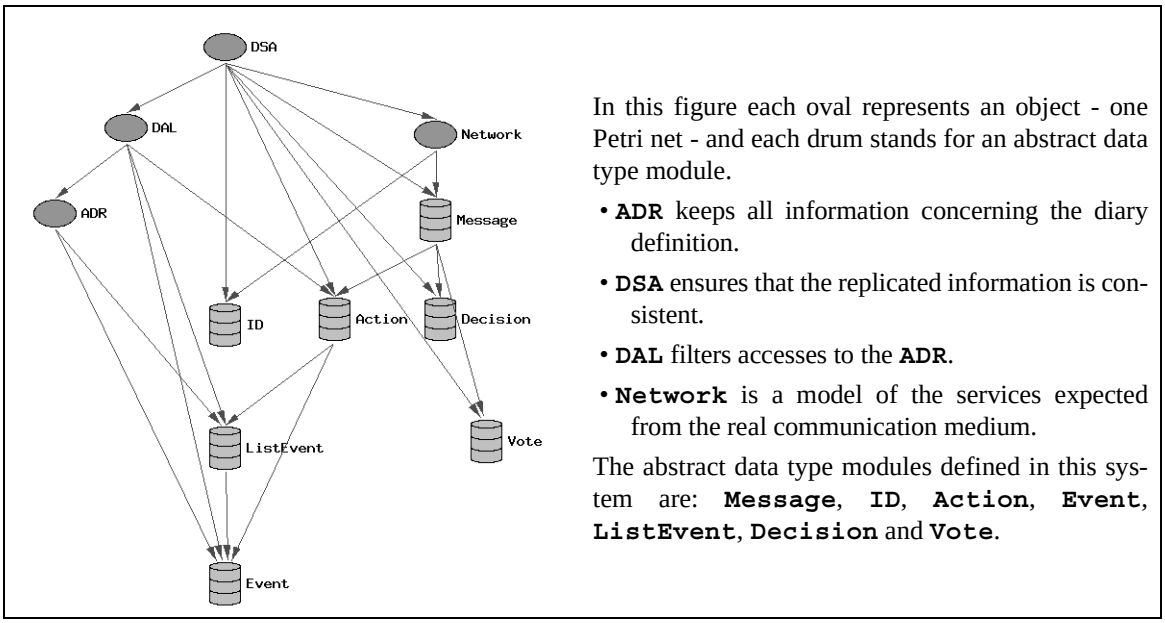


Figure 14. Module Enrichment Relationships of the Diary Specification

For our example we define a simple collaborative diary, with a replicated architecture, for members of a ‘software engineering laboratory’. This diary assists in the management of meetings in the laboratory conference room by letting several users view and modify its contents simultaneously from their individual workstations, while preventing conflicts, i.e. planning of overlapping events. The diary allows the following services: Consultation, addition, cancellation and replacement of events. An event structure is simply composed of a day, a beginning and ending time and a comment line. Each site has four components for the diary of each member: the Abstract Document Representation (**ADR**), the Distributed Synchronization Algorithm (**DSA**), the Data Access Layer (**DAL**) and the Graphical Interface Layer (not represented in figure 14). This architecture is inspired from [Karsenty 96].

The **DSA** is to guarantee consistency between the replicas of the diary. To this end, we rely upon a very simple mechanism: When a user modifies its copy of the diary, the corresponding event modification must be accepted by the other participants. If by misfortune another user wants to validate at the same moment a conflicting event, then the respective **DSA**s will detect the problem and reject both updates. We will not bother specifying here how the conflict is actually solved: Let us just imagine that the interface signals the error, and that the humans are able to communicate by other means to arrive at an agreement.

From an algorithmic point of view, the validation of an event is implemented by an atomic three step operation: First broadcast(1) the wish to validate an event, and then wait for all participants to acknowledge by a vote. If a single answer is negative then broadcast an

abort, otherwise a commit order. This algorithm is called the *two-phase commit* protocol (2PC) [Gray 78]: The first phase consists of the two first steps above, and the second phase is the communication of the final decision.

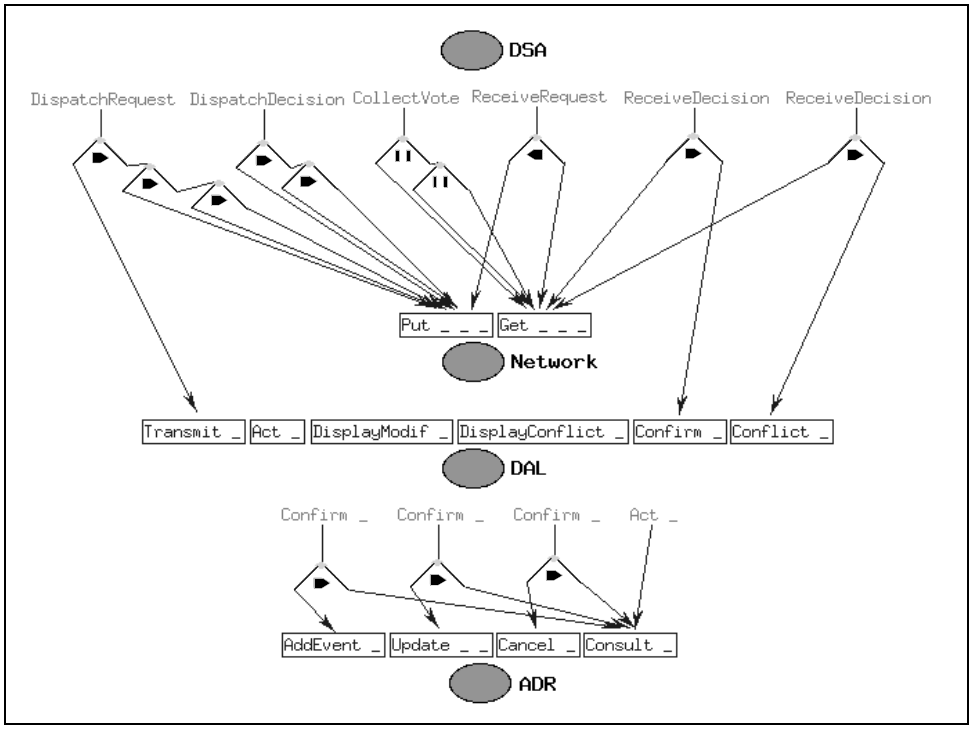


Figure 15. Global View of the Synchronizations in the Control of a Collaborative Diary

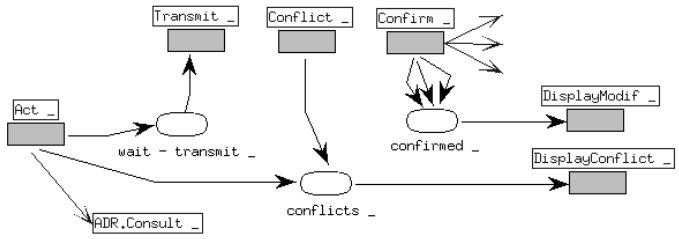
According to our terminology, the events which represent the evolution of the net can be of two types: *methods* or *internal transitions*. The methods of an object are activated externally while internal transitions are activated locally, as soon as their conditions are validated. The events modify the state of the system and the symbolic links between the objects express synchronization constraints between the events of these objects. The constraints are described by a combination of sequential (graphically ► or ◄, textually "**..**"), simultaneous (**||**, respectively "**&**"), alternative (**+** and "**+**") or empty synchronization operators.

1. Here a broadcast includes for simplicity the sender in the list of addressees, hence *n* messages are sent instead of *n*-1, where *n* is the number of users of the distributed diary.

In summary, our formalism is based on the following components:

- Algebraic abstract data types
- Petri nets
- Modularity and synchronizations

From a graphical point of view, a system can be seen at two different levels. The first level is an abstract view of the application, which represents either the objects with their synchronization links (figure 15) or the module enrichment - or clientship - relations (figure 14). The second level represents the algebraic Petri net within each object and displays its places, transitions and methods with parameters (Figure 16, which is equivalent to figure 17). A method is the entry point for synchronizations originating from other objects.



This is the graphical representation of the behavior of object **DAL**:

- Method **Act** is offered to the clients of the diary in order to validate an **action**, i.e. a modification to the current list of events. If a conflict can be detected locally, the action is directly inserted into the **conflicts** place. Otherwise, the action is transmitted to the place **wait-transmit**, so that the **DSA** object may, by calling the method **Transmit**, further validate it globally on the network.
- Method **Conflict** allows the **DSA** to inform of actions that conflict with the other participants of the diary.
- Method **Confirm** allows the **DSA** to provide actions that have been globally validated on the network.
- Methods **DisplayModif** and **DisplayConflict** are to tell the graphical interface when new events are available for displaying.

Figure 16. An algebraic Petri net: Internal view of object **DAL**

Figure 17 (below) is the textual specification corresponding to the graphical representation shown in figure 16. Only the behavior of method `Act` and one axiom of method `Confirm` is

listed here. Notice that CO-OPN is very liberal about the usage of parenthesis. Operands may also be placed freely within or around their "operators" (generators, operations, methods or places): This is sometimes called *mixfix* or *distributed* notation.

```
OBJECT DAL;
INTERFACE
  USE ADR, ListEvent, Event, Action;
  METHODS
    Act _              : action;
    DisplayModif _     : listevent;
    DisplayConflict _: action;
    Transmit _         : action;
    Confirm _          : action;
    Conflict _         : action;
BODY
  PLACES
    confirmed _        : listevent;
    conflicts _        : action;
    wait-transmit _    : action;
  AXIOMS
    ActOk :: Consistent(a,l)=true =>
                  Act a WITH Consult(l) : -> wait-transmit a;
    ActNotOk :: Consistent(a,l)=false =>
                  Act a WITH Consult(l) : -> conflicts a;
    CnfAddEv :: Confirm (AddEvent(e)) WITH AddEvent(e) .. Consult(l)
                  : -> confirmed l;
    ...
  WHERE
    a : action;
    l : listevent;
END DAL;
```

Figure 17. Partial Specification of Object **DAL**

In the interface of the object, the **USE** list reports the other objects with which we perform synchronizations (`ADR`) and the Adts (`ListEvent`, `Event`, `Action`) needed in order to declare the profiles of the exported methods (`Act`, `DisplayModif`, `DisplayConflict`, `Transmit`, `Confirm` and `Conflict`). Each underscore character ('_') plays the role of a place holder for an argument.

In the **BODY** of the **DAL** object, the **WHERE** keyword introduces the local variable declarations. Although strongly typed, they should be considered as logic variables in the sense of the Prolog programming language [Colmerauer 83], since they operate by unification instead of destructive assignment.

The concrete syntax of the axiom describing an event, be it a method or a transition, is:

```
[ [ AxiomName "::" ] AxiomCondition "=>" ]
  EventNameAndParams [ "WITH" AxiomSynchronization ] ":"
    [ AxiomPrecondition ] "->" [ AxiomPostcondition ] ";"
```

The `AxiomName` part is completely optional, but may be helpful for locating errors during compilation, for commenting the code, or, as we will see later, for clearly labelling the non-deterministic choices according to the different axioms of the event. In our example, `ActOk` and `ActNotOk` denote the cases for method `Act` where the action `a` to validate is locally acceptable, respectively conflicts with the local view of the event list `l`.

`AxiomCondition` is a conjunction of general relations over the variables which we cannot or do not want to state (by direct unification) in any other part of the axiom. The expression `Consistent(a,l)=true` is an instance of such a condition.

`EventNameAndParams` tells which specific event the axiom defines, as well as the names of its formal parameters in the case where the event denotes a method.

The synchronization expression is given in `AxiomSynchronization`. It states the names of the methods which are called for establishing rendez-vous, and the terms through which data are exchanged. Notice that the syntax of CO-OPN does not indicate the modes of the parameters, i.e. whether they are read or written to: This characteristic is related to the operational view of the language which is again similar to Prolog, and is one of the mechanisms which contribute to maintaining a high level of abstraction in the specifications.

The `AxiomPrecondition` and `AxiomPostcondition` parts establish the links to the input and output places of the event in the sense of Petri nets, and state with algebraic expressions the additional conditions or transformations to establish on the tokens.

The source specification of Adt **`Event`** is listed below (figure 18). It must be noted that we do not provide any graphical representation for the internal structure of Adts.

```
ADT Event;
INTERFACE
  (: Import standard modules :)
  USE Booleans, String, Time;
  SORT event;
  GENERATORS
    < _ _ _ _ _> : date, daytime, daytime, string -> event;
  OPERATIONS
    _ = _             : event event -> boolean;
    Overlapping _ _   : event event -> boolean;
BODY
  AXIOMS
    (: Define equality :)
    (< day1, start1, end1, cmnt1 > = < day2, start2, end2, cmnt2 >) =
      (day1 = day2 AND start1 = start2 AND end1 = end2 AND cmnt1 = cmnt2);

    (: Define 'Overlapping' working hours :)
    Overlapping   < day1, start1, end1, cmnt1 >
                  < day2, start2, end2, cmnt2 > =
      (day1 = day2 AND start1 <= end2 AND end1 > start2);

  WHERE
    day1, day2 : date;
    start1, start2 : daytime;
    end1, end2 : daytime;
    cmnt1, cmnt2 : string;
END Event;
```

Figure 18. Specification of Adt **`Event`**

The syntactical structure of an Adt module is very similar to the one of object modules. In the interface, the difference is that the former declares *sorts* (or types), *generators* (akin to constructors in programming languages) and *operations* (the same as functions) where the latter has *methods*. In the body, an Adt has only axioms and variable declarations(2).

The syntax of a generator or operation axiom is as follows:

```
[ [ AxiomName "::" ] AxiomCondition "=>" ]
  GenOpNameAndParams "=" EquationRightHand ";"
```

The `AxiomName` and `AxiomCondition` parts are both optional and play here the same roles as in event axioms. In contrast, it can be mentioned that AADTs do not exhibit comparable levels of non-determinism as CO-OPN objects, since they provide only deterministic operations and immutable values (once created, they are not sensitive to their environment).

The mandatory part of the axiom is an equation. Its left-hand side, `GenOpNameAndParams`, tells which generator or operation the axiom defines, as well as its formal parameters. The

2. In fact it may also contain *theorems*, which are formulas similar to axioms, except that they are considered as comments since they express properties that are not intended to be interpreted.

right-hand side of the equation (`EquationRightHand`) is an expression and will be interpreted in this report in a rewrite approach, i.e. as the result of the generator or operation. This subject needs however further clarifications and will be postponed until chapter 4, which gives a more formal presentation of the interpretation and compilation of algebraic specifications.

3.4 CO-OPN Syntax

The purpose of this section is to describe the abstract syntax of the CO-OPN formalism. The concrete syntax will be referred to as seldom as possible, since we are more interested in the semantic aspects of the language.

Recall that a CO-OPN specification is composed of two kinds of descriptions associated to two kinds of modules: *Adt modules* and *Object modules*. Adt modules are used to describe the algebraic abstract data types involved in a CO-OPN specification, while Object modules correspond to the description of the individual algebraic Petri nets.

Throughout this paper we consider a universe which includes the disjoint sets: **O**, **S**, **F**, **M**, **P**, **X**, **N**. These sets correspond, respectively, to the sets of all objects, sorts, functions, methods, places, variables and axiom names. The "S-sorted" notation facilitates the subsequent development. Let S be a set, then a S-sorted set A is a family of sets indexed by S, and we write $A=(A_s)_{s\in S}$. A S-sorted set of disjoint sets of variables is called a *S-sorted variable set*. Given two *S-Sorted sets* A and B, a *S-sorted function* $\mu : A \rightarrow B$ is a family of functions indexed by S denoted $\mu=(\mu_s : A_s \rightarrow B_s)_{s\in S}$.

3.4.1 Signature and Interface

As usual, a signature groups two elements of an algebraic abstract data type, i.e. a set of sorts and some operations. However, in the context of structured specifications, a signature can potentially use elements not locally defined, i.e. defined outside the signature itself; otherwise the signature is said *complete*. Notice that in the next definition the profile of the operations are defined on the set of all sort names **S**.

Definition 1: *Adt Signature*

An *Adt signature* is a couple $\Sigma = < S,F >$ in which

- S is a set of sort names of **S**;
- $F = (F_{w,s})_{w \in \mathbf{S}^*, s \in \mathbf{S}}$ is a $(\mathbf{S}^* \times \mathbf{S})$-sorted set of function names of **F**, where $\mathbf{S}^*$ denotes the sequences of sorts in **S**.

◊

We often denote a function name $f \in F_{s_1 \ldots s_n, s}$ by $f : s_1,\ldots,s_n \rightarrow s$, and a constant $f \in F_{\varepsilon,s}$ by $f : \rightarrow s$ (ε being the empty sequence).

Definition 2: *Constructors and Operations*

Inside F we can distinguish C and OP, respectively a finite set of *constructors*, also called *generators*, and a finite set of *operations*, sometimes called *defined* or *derived operations*. The sets C and OP are disjoint. Moreover we have $F = C \cup OP$. ◊

The distinction between generators and operations is needed for the interpretation of AADTs within the frame of rewrite systems (chapter 4). At a more abstract level, these two notions can be assimilated.

Similarly to the notion of Adt signature, the elements of an Object module which can be used from the outside are grouped into an Object interface. The Object interface of an Object module includes the identifier of the Object and the set of methods which correspond to the services provided by the object.

Definition 3: *Object Interface*

An *Object interface* is a couple $\Omega = < \{o\}, M >$ in which

- $o \in \mathbf{O}$ is the identifier of an Object module;
- $M = (M_{o,w})_{w \in \mathbf{S}^*}$ is a finite $\mathbf{S}^*$-sorted set of method names of $\mathbf{M}$.

◊

On this basis we will be able to build a global signature and a global interface.

Definition 4: *Global Signature and Global Interface*

Let $\overline{\Sigma} = (\Sigma_i)_{1 \le i \le n}$ be a set of Adt signatures and $\overline{\Omega} = (\Omega_j)_{1 \le j \le m}$ be a set of Object interfaces such that $\Sigma_i = < S_i, F_i >$ and $\Omega_j = < \{o_j\}, M_j >$.

The *global signature* is: $\Sigma_{\overline{\Sigma}} = \langle \bigcup_{1 \le i \le n} S_i \ , \ \bigcup_{1 \le i \le n} F_i \rangle$

The *global interface* is: $\Omega_{\Omega} = \langle \bigcup_{1 \le i \le n} \{o_j\}, \bigcup_{1 \le i \le n} M_j \rangle$ ◊

3.4.2 Variables, Terms and Equations

As usual the properties of the operations of an AADT are described by means of equations (more generally conditional positive equations) which consist of pairs of terms. The set of terms of an AADT corresponds to the correct expressions of that type that can be written.

To describe properties that are as general as possible, we need beforehand the notion of variable.

Definition 5: *S-Sorted Variable Set*

Let $\Sigma = < S, F >$ be a complete signature. An S-sorted set of Σ-variables is an S-indexed set $X = (X_s)_{s \in S}$ of disjoint subsets of $\mathbf{X}$. ◊

A Σ-variable is usually simply called a variable.

Definition 6: *Set of all Terms*

Let $\Sigma = < S, F >$ be a complete signature and X be an S-sorted variable set. The *set of all terms* over Σ and X is the S-sorted set $T_{\Sigma,X} = ((T_{\Sigma,X})_s)_{s \in S}$ inductively defined as:

- $x \in (T_{\Sigma,X})_s \; \forall \; x \in X_s$;
- $f \in (T_{\Sigma,X})_s \; \forall \; f : \rightarrow s \in F$;
- $f(t_1,...,t_n) \in (T_{\Sigma,X})_s \; \forall \; f : s_1,...,s_n \rightarrow s \in F$ and $\forall \; t_i \in (T_{\Sigma, X})_{s_i}$ $(1 \leq i \leq n)$.

◊

$T_{\Sigma,X}$ is simply written T_Σ when $X=\emptyset$.

Definition 7: *Variables, Groundness and Linearity of Terms*

Let $\Sigma = < S, F >$ be a complete signature and X be an S-sorted variable set. $Vars(t)$ is the set of variables occurring in the term $t \in (T_{\Sigma,X})_s$. When $Vars(t)=\emptyset$, the term t is said *ground*, and when each variable is present no more than once, t is said *linear.* ◊

Definition 8: *Equation and Positive Conditional Equation*

Let $\Sigma = < S, F >$ be a complete signature and X be an S-sorted variable set. An *equation* is a pair $< t, t' >$ of equally sorted terms:$\exists$ a sort s such that $t,t' \in (T_{\Sigma,X})_s$. A *positive conditional equation* is an expression $e_1 \wedge ... \wedge e_n \Rightarrow e$ where e,e_i $(1 \leq i \leq n)$ are equations. ◊

3.4.3 Adt Module

An Adt module consists of an Adt signature which may use elements not locally defined, a set of positive conditional equations, and some variables.

Definition 9: *Adt Module*

Let Σ be a set of Adt signatures and Ω be a set of Object interfaces such that the global $\Sigma_{\Sigma,\Omega} = < S, F >$ is complete. An *Adt module* is a triplet $AM_{\Sigma,\Omega} = < \Sigma,X,\Phi >$ in which

- Σ is an Adt signature;

- $X = (X_s)_{s \in S}$ is an S-sorted variable set;
- Φ a set of positive conditional equations over $\Sigma_{\Sigma,\Omega}$ and X.

◊

In the concrete syntax of CO-OPN, it is possible to associate a name $n \in \mathbf{N}$ to each axiom. The name n must be unique within the defining module. An axiom name has no semantic meaning: It constitutes a form of comment.

3.4.4 Multi-Sets

Algebraic nets use the notion of multi-set so as to have indistinguishable copies of independent values. Thus, in order to express terms over multi-sets, we define now the multi-set extension of a signature. Formally a multi-set over a set E is a mapping from E to $\mathbb{N}$. The set of all multi-sets over a set E is defined by the set of all functions $[E] = \{ f \mid f : E \to \mathbb{N}\}$ equipped with the following operations:

$$[e]^{[E]}(e') = \begin{cases} 1 \text{ if } e = e' \\ 0 \text{ otherwise} \end{cases} \quad \text{for all } e, e' \in E;$$

$$(f +^{[E]} g)(e) = f(e) + g(e) \quad \text{for all } f, g \in [E] \text{ and for all } e \in E.$$

We note $\varnothing^{[E]}$ the element of $[E]$ such that $\varnothing^{[E]}(e)=0$ for all $e \in E$.

The multi-set extension of a given Adt signature consists of the signature, enriched for each sort, with the multi-set sort, and the multi-set operations ∅ (empty set), [_] (coercion to the single-element set) and + (set union).

Definition 10: *Multi-Set Extension of a Signature*

Let $\Sigma = < S, F >$ be a signature. The multi-set extension of Σ is

$$[\Sigma] = \langle S \cup \bigcup_{s \in S} \{[s]\}, F \cup \bigcup_{s \in S} \left\{ \begin{array}{l} \varnothing_s : \ \to [s], \\ [\_]_s : s \to [s], \\ +_s : [s], [s] \to [s] \end{array} \right\} \rangle . \qquad \text{◊}$$

3.5 CO-OPN Objects and Synchronizations

The main problem one has to face when modelling systems by the means of Petri nets is their lack of scalability: They are unusable when facing consequential projects. This is why CO-OPN includes the notion of object. As seen in section 3.3, an object is an algebraic Petri

net exporting transitions with parameters (called *methods*) to the outside world. Objects may thus synchronize and communicate.

Objects encapsulate their internal behaviour by hiding the corresponding declarations inside the body section. As suggested by the example of figure 21, the hidden entities are:

- the places and their initial markings
- the transitions (sometimes called *internal transitions* by contrast with methods)
- the axioms describing the behaviour
- the variables of the axioms

We can now proceed to the formal definitions for the description of CO-OPN objects.

3.5.1 Behavioral Axioms

Before defining what a behavioral axiom is, let us precise that our formalism provides two different categories of events: The *invisible* and the *observable* events, both of which can involve an optional *synchronization expression*. The invisible events describe the spontaneous reactions of an object to some stimuli. They correspond to the internal transitions which are denoted by τ and not by a specific name as in the concrete CO-OPN language. As for the observable events, they correspond to the methods which are accessible from the outside. A synchronization expression offers to an object the means of choosing how he wishes to synchronize with other partners (or himself). Three synchronization operators are provided: '**&**' for *simultaneity*, '**..**' for *sequence*, and '**+**' for *alternative*. In order to designate a particular method *m* of a given object *o*, the usual dot notation has been adopted(3).

The set of all events over a set of parameter values A_S, a set of methods *M*, and a set of object identifiers *O* is written $\mathbf{E}_{A,M,O}$. Because this set is used for various purposes, we give here a generic definition on A.

Definition 11: *Generic Set of all Events*

Let *S* be a set of sorts and S^* the sequences of sorts in *S*. Let us consider A_s a set of terms of sorts $s \in S$, *O* a set of object identifiers and $M_{o,w}$ a set of methods indexed by their defining object $o \in O$ and arity *w* in S^*. Atomic events *Event* of $\mathbf{E}_{A,M,O}$ are built from this syntax:

3. This notation does not correspond to the concrete syntax of CO-OPN v1.5, which uses instead the keyword **IN** as follows: *m* **IN** *o* (*arguments of m*).

Event	→	*Invisible* | *Observable*
Internal	→	$o.\tau$
Invisible	→	*Internal*	*Internal* **WITH** *Synchronization*
Invocation	→	$o.m\ (a_1,...,a_n)$
Observable	→	*Invocation*	*Invocation* **WITH** *Synchronization*
Synchronization	→	*Invocation*	*Synchronization* **&** *Synchronization* | *Synchronization* **..** *Synchronization* | *Synchronization* **+** *Synchronization*

where $s_i \in S\ (1 \leq i \leq n)$, $a_1,...,a_n \in A_{s_1} \times ... \times A_{s_n}$, $m \in M_{o, s_1...s_n}$, and $o \in O$. ◊

For example, the observable event "$o.m(a_1,a_2)$ **WITH** $o_1.m_1(a_1)$ & $o_2.m_2(a_2)$" represents the simultaneous synchronization of the method m of an object o with both the methods m_1 and m_2 of two objects o_1 and o_2.

Synchronization expressions state relations between modules: Events should be considered as predicates to be proved or refutated with respect to the current state of the object system by a hidden search mechanism, the resulting behavior being the activation of the satisfying transitions and methods, as in more conventional schemes. In [Buchs&Guelfi 91] terms built from the non-terminal symbol *Synchronization* were of sort *Rendez-Vous*, which may give another insight into their intended signification. This also suggests that method invocations are *synchronous*: The caller[4] is blocked until the receipt of the reply, *success* or *failure*. These remarks will be confirmed by the inference rules given later in this chapter.

It is interesting to see that, as an extension of simple transitions of classical Petri nets, CO-OPN events are also considered as atomic, i.e. indivisible, actions. This is even true for events with arbitrarily complex synchronization expressions built from the given syntax.

Now we give the definition of the behavioral axioms which are used to describe the properties of observable and invisible events (respectively, methods and internal transitions).

4. We do not specify formally what characterizes a caller: Intuitively, we can consider that it consists of a thread of control in the Object which emits the invocation (simultaneity may induce intra-object concurrency).

Definition 12: *Behavioral Axiom*

Let $\Sigma = < S,OP >$ be a complete Adt signature. For a given S^*-sorted set of methods M, a set of Object identifiers O, an S-sorted set of places P, and an S-sorted set of variables X. A *behavioral axiom* is an expression

$$Cond \Rightarrow Event \textbf{ : } Pre \rightarrow Post \textbf{ ;}$$

defined as follows:

- $Event \in \mathbf{E}_{(T_{\Sigma,X}),\, M,\, O}$;
- *Cond* is a set noted $e_1 \wedge ... \wedge e_n$ where all e_i $(1 \leq i \leq n)$ are equations over Σ and X;
- $Pre = (Pre_p)_{p \in P}$ is a P-indexed family of terms over $[\Sigma]$ and X s.t. $(\forall\ s \in S)\ (\forall\ p \in P_s)\ (Pre_p \in (T_{[\Sigma],X})_{[s]})$;
- $Post = (Post_p)_{p \in P}$ is a P-indexed family of terms over $[\Sigma]$ and X s.t. $(\forall\ s \in S)\ (\forall\ p \in P_s)\ (Post_p \in (T_{[\Sigma],X})_{[s]})$.

The set of terms *Pre* and *Post* are called respectively the *precondition* and the *postcondition* of the event. The expression *Cond* is the *global condition*[5] over the event. ◊

As with axioms of Adt modules, the concrete syntax of CO-OPN also allows giving names to behavioral axioms:

$$Name \textbf{ :: } Cond \Rightarrow Event \textbf{ : } Pre \rightarrow Post \textbf{ ;}$$

The $Name \in \mathbf{N}$ of a behavioural axiom does not have any semantic meaning: It is optional and only serves as a kind of comment. It is however required that all axiom names be distinct within their module, which also determine their lexical scope.

As for operations of AADTs, events of an object may be defined by several axioms, corresponding to different, not necessarily mutually exclusive, situations. This suggests that the name of a transition τ or a method m actually corresponds to a *familiy* of events, and not to a simple event, when compared with classical algebraic Petri nets. For instance, when an object o_1 makes an invocation to method m of object o_2, it does not "know" which axiom will actually be selected in o_2 to enable m. On the other hand, internal transitions are never explicitly called, so when several axioms define a transition of name τ, it is merely meant as an abstraction mechanism for grouping together a set of logically related elementary transitions.

5. This terminology is non-standard, but we prefer to reserve the word *condition* for a generic usage referring to both the precondition and the global condition. We justify the qualifier *global* by the fact that *Cond* may express conditions over all variables of an axiom whether they come from *Event* or *Pre*.

3.5.2 Object Module

The purpose of an Object module is to describe an encapsulated algebraic net. It consists of an interface, the state of the object (its set of places), some variables as well as a set of behavioral axioms which describe the properties of the methods and of the internal transitions.

Definition 13: *Object Module*

Let Σ be a set of Adt signatures, Ω be a set of Object interfaces. An *Object module* is a 4-tuple $OM_{\Sigma, \Omega} = \langle \Omega, P, X, \Psi \rangle$ in which

- $\Omega = < \{o\}, M >$ is an Object interface;
- $P = (P_s)_{s \in S}$ is a finite S-sorted set of place names of **P**;
- $X = (X_s)_{s \in S}$ is a S-sorted variable set of **X**;
- Ψ is a set of behavioral axioms over $\Sigma_{\Sigma,\Omega}$, M, P, $\{o\}$, and X.

◊

3.5.3 N-tuples as Tokens and Net Inscriptions

It is often convenient to structure the tokens and conditional terms on the arcs as cross-products of existing sorts without having to define explicitely new AADTs for this purpose. This extension is straightforward and does not constitute any problem on the formal side [Reisig 91]. In the collaborative diary example, the object **`Network`** has a place `Channel` which represents the communication medium: The tokens it contains is a cross-product of sort `message` - the useful data - and sort `id` - the emitter and addressee identities - built without having to marshal the information inside specific packet types. Figure 19 shows how this structuring facility is written in CO-OPN.

```
OBJECT Network;
INTERFACE
  USE Message, ID;
  METHODS
    Put _ _ _ : message id id;
    Get _ _ _ : message id id;
BODY
  PLACES
    Channel _ _ _ : message id id;
  AXIOMS
    Put msg iddest idorigin : -> Channel msg iddest idorigin;
    Get msg iddest idorigin : Channel msg iddest idorigin ->;
  WHERE
    msg : message;
    iddest, idorigin : id;
END Network;
```

Figure 19. CO-OPN Object as Abstraction of the Network

3.5.4 CO-OPN Specification

At last, a CO-OPN specification is a collection of Adt and Object modules.

Definition 14: *CO-OPN Specification*

Let Σ be a set of Adt signatures, Ω be a set of Object interfaces. A *specification* consists of a set of Adt and Object modules:

$$Spec_{\Sigma,\Omega} = \left\{ \left(AM_{\Sigma,\Omega} \right)_i \;\middle|\; (1 \le i \le n) \right\} \cup \left\{ \left(OM_{\Sigma,\Omega} \right)_j \;\middle|\; (1 \le j \le m) \right\}.$$

We denote a specification $Spec_{\Sigma,\Omega}$ by *Spec* when Σ and Ω are, respectively, included in the global signature and in the global interface of the specification. In this case *Spec* is said *complete*. ◊

From a specification *Spec* two dependency graphs can be constructed. The first one consists of the dependencies within the algebraic part of the specification, i.e. between the various Adt signatures. The second dependency graph corresponds to the clientship relations between the Object modules. Both of these graphs consist of the specification *Spec* and of a binary relation over *Spec* denoted *AD* for the algebraic dependency graph, and *OD* for the Object clientship dependency graph. The relation *AD* is constructed as follows: For any pair of modules *Md,Md'* of *Spec*, < *Md,Md'* > is in *OD* iff the Adt module or the Adt signature induced by *Md* uses something defined in the Adt signature of *Md'* or in the Adt signature induced by *Md'*. As for the relation *OD*, for any Object modules *Md,Md'*, then < *Md,Md'* > is in *OD* iff there is a synchronization expression of a behavioral axiom of *Md* which involves a method of *Md'*.

Thus, a well-formed specification is a specification with two constraints on the dependencies between the modules composing the specification. These hierarchical constraints are necessary in the theory of algebraic specifications and in the Object module dimension of our formalism as it will be shown in the next section.

Definition 15: *Well-formed Specification*

A complete specification *Spec* is *well-formed* iff

i) the algebraic dependency graph < *Spec,AD* > has no cycle;
ii) the object clientship dependency graph < *Spec,OD* > has no cycle.

◊

3.6 CO-OPN Semantics

This section presents the semantic aspects of the CO-OPN formalism which are mainly based on the notions of algebraic specifications and transition systems. First we briefly recall some basic definitions in relation with the semantics of algebraic specifications and their multi-set extension (we refer to [Ehrig&Mahr 85] for more detailed descriptions). Afterward we present all the inference rules which construct the semantics of a CO-OPN specification in terms of transition systems. For this description we base ourselves on the original CO-OPN paper [Buchs&Guelfi 91] and also partly on [Biberstein 97]. We completed this work with numerous examples and comments in order to orient the discourse in the direction of the algorithmic approach of chapter 6.

3.6.1 Algebras and Multi-set Extension

Let *Spec* be a well-formed CO-OPN specification, and $Spec^A = < \Sigma, X, \Phi >$ be its associated algebraic specification in which $\Sigma = < S, F >$. A Σ-algebra A consists of a S-sorted set $A = (A_s)_{s \in S}$ and a $(S^* \times S)$-sorted family of total functions $F^A = \left(f^A_{s_1 \ldots s_n, s} \right)_{f: s_1 \ldots s_n, s \to s \in F}$.

An *assignment* is an S-sorted function $\sigma = (\sigma_s : X_s \to A_s)_{s \in S}$. An *interpretation* of terms of $T_{\Sigma,X}$ in A is an S-sorted function $[\![\_]\!]^\sigma = \left([\![\_]\!]^\sigma_s : (T_{\Sigma, X})_s \to A_s \right)_{s \in S}$ such that:

i) $[\![x]\!]^\sigma_s \stackrel{\text{def}}{=} \sigma_s(x)$ if $x \in X_s$;
ii) $[\![f]\!]^\sigma_s \stackrel{\text{def}}{=} f^A_s$ if $f \in F_{\varepsilon,s}$;
iii) $[\![f]\!]^\sigma_s (t_1, \ldots, t_n) \stackrel{\text{def}}{=} f^A_s ([\![t_1]\!]^\sigma_{s_1}, \ldots, [\![t_n]\!]^\sigma_{s_n})$ if $f \in F_{s_1 \ldots s_n, s}$.

When it is necessary to specify the Σ-algebra, we write $[\![\_]\!]^\sigma_A$.

A Σ-algebra A satisfies an equation $< t, t' >$ for an assignment $\sigma : X \to A$ iff $[\![t]\!]^\sigma = [\![t']\!]^\sigma$, and we note this $A,\sigma \models < t, t' >$. We also have $A,\sigma \models ((\bigwedge_{1 \le i \le n} \phi_i) \Rightarrow \phi')$ iff $(\forall i)$ $A,\sigma \models \phi_i$ implies $A,\sigma \models \phi'$. A positive conditional equation ϕ is *valid* in a Σ-algebra A iff $A,\sigma \models \phi$ for any assignment σ, and we note this $A \models \phi$. An algebraic specification $Spec^A$ is valid in a Σ-algebra A iff all the conditional positive equations of $Spec^A$ are valid in A. A *model* of $Spec^A$ is a Σ-algebra in which $Spec^A$ is valid. The set of models of $Spec^A$ is a sub-class of $Alg(\Sigma)$, which is noted $Mod(Spec^A)$.

In order to perform the semantic multi-set extension of a Σ-algebra A we directly provide the semantics of multi-sets. From section 3.4.4 the semantic extension of a Σ-algebra A is

$$[A] = \langle A \cup \left(\bigcup_{s \in S} [A_s] \right), F^A \cup \left(\bigcup_{s \in S} \varnothing^{[A_s]}, [\_]^{[A_s]}, +^{[A_s]} \right) \rangle .$$

The multi-set syntactic extension of an algebraic specification $Spec^{A} = < \Sigma,X,\Phi >$ is noted $[Spec^{A}] = < [\Sigma],X,\Phi >$. We restrict the set of models of $[Spec^{A}]$ to $Mod([Spec^{A}]) \stackrel{def}{=} \{[A] \mid A \in Mod(Spec^{A})\}$.

3.6.2 Object States

The state of an object is based on the notion of marking, i.e. a mapping which returns a multi-set of algebraic values for each place of the object. Some authors chose to consider each place as equivalent to a variable at the programming language level [Guelfi 94]. We prefer however to view each individual token as a variable, and the place as a concurrency control mechanism for accessing its variables. Although subtle in practice, we think that this distinction is more consistent with the facility of algebraic nets for structuring the tokens, and with the synchronization rules of CO-OPN (to be presented later in this chapter), which imply a notion of mutual exclusion at the token level between concurrent method invocations.

For a given CO-OPN specification *Spec* and its set of objects *O*, we define a marking *m* as follows.

Definition 16: *Object Marking*

Let be a set of sorts *S* and an *S*-sorted set of places *P*. A *marking m* is defined as $m : P \rightarrow [A]$ such that $p \in P_s$ implies $m(p) \in A_{[s]}$. We denote by $Mark_{Spec,A}$ the set of all markings and by $InitMark_{Spec,A}$ the initial marking. ◊

Definition 17: *Transition System*

Let *Spec* be a CO-OPN specification and *A* be a model of $Spec^{A}$. A *transition system* is defined as

$$TS_{Spec,A} \subseteq Mark_{Spec,A} \times \mathbf{E}_{A,M(Spec),O} \times Mark_{Spec,A}$$

We write $m \xrightarrow{e} m'$ the step *e* from a *source* state *m* to a *target* state *m'*, where $m,m' \in Mark_{Spec,A}$ and $e \in \mathbf{E}_{A,M(Spec),O}$. ◊

Definition 18: *Marking Domain*

Let *Spec* be a CO-OPN specification and *A* be a model of $Spec^{A}$. The *marking domain* of a marking $m \in Mark_{Spec,A}$ is given by

$$Dom(m) = \{ p \mid m(p) \text{ is defined}, p \in \mathbf{P} \}$$ ◊

Remark that if $Dom(m) = \emptyset$ then we denote *m* by $\perp$.

Let us introduce now two basic operators on markings: The first one, $m_1 \hat{=} m_2$, is a predicate which determines if two markings m_1 and m_2 are equal on their common places, and the second one, $m_1 \overrightarrow{\cup} m_2$, considers two markings and returns a marking with the values of the marking m_1 plus the values of the places of m_2 which are not present in m_1 (the arrow symbolizes the assymmetry of the operation: m_1 partly "overwrites" m_2).

Definition 19: *Common Markings, Marking Fusion*

Let *Spec* be a CO-OPN specification and *A* be a model of $Spec^A$. For two markings $m_1, m_2 \in Mark_{Spec,A}$:

$$m_1 \hat{=} m_2 \Leftrightarrow m_1(p) = m_2(p)\ \forall\ p \in Dom(m_1) \cap Dom(m_2)$$

$$m_1 \overrightarrow{\cup} m_2 = m_3 \Leftrightarrow$$

$$\forall\ p \in Dom(m_1) \cup Dom(m_2),\ m_3(p) = \begin{cases} m_1(p) \text{ if } p \in Dom(m_1) \\ m_2(p) \text{ if } p \in Dom(m_2) \backslash Dom(m_1) \\ \text{undefined otherwise} \end{cases}$$

◊

Theorem 1: *Marking Fusion is Associative*

Let *Spec* be a CO-OPN specification and *A* be a model of $Spec^A$. We have, $\forall\ m_1, m_2, m_3 \in Mark_{Spec,A}$:

$$\left(m_1 \overrightarrow{\cup} m_2\right) \overrightarrow{\cup} m_3 = m_1 \overrightarrow{\cup} \left(m_2 \overrightarrow{\cup} m_3\right)$$

Proof: We simply calculate successively both sides of the equation.
$\forall\ p \in Dom(m_1) \cup Dom(m_2) \cup Dom(m_3)$

$$\left(m_1 \overrightarrow{\cup} m_2\right) \overrightarrow{\cup} m_3 = \begin{cases} m_1(p) \text{ if } p \in Dom(m_1) \\ m_2(p) \text{ if } p \in Dom(m_2) \backslash Dom(m_1) \\ m_3(p) \text{ if } p \in Dom(m_3) \backslash (Dom(m_2) \backslash Dom(m_1)) \\ \text{undefined otherwise} \end{cases}$$

$$m_1 \overrightarrow{\cup} \left(m_2 \overrightarrow{\cup} m_3\right) = \begin{cases} m_1(p) \text{ if } p \in Dom(m_1) \\ m_2(p) \text{ if } p \in Dom(m_2) \backslash Dom(m_1) \\ m_3(p) \text{ if } p \in (Dom(m_3) \backslash Dom(m_2)) \backslash Dom(m_1) \\ \text{undefined otherwise} \end{cases}$$

The equation holds because of the associativity of the set difference (operator '\'). ◊

3.6.3 Inference Rules

In order to construct the semantics, defined mainly by two concurrent transition systems, of a CO-OPN specification, we provide a set of inference rules. These rules are expressed as *structured operational semantics* [Plotkin 77], i.e. they have the form

$$\frac{t_1, t_2, \ldots, t_n}{t}$$

where $t,t_1,t_2,...,t_n \in TS_{Spec,A}$. The steps $t_1,t_2,...,t_n$ above the line are called the *premises* and the step t below the line is called the *conclusion*. The meaning of such a rule is that the step in the conclusion may be done if all steps in the premises can take place.

The first transition system $\rightarrow$ is used to compute the behavior in unstable object states, and the second $*\rightarrow$ to describe only the events observable after stabilization. This stabilization process calculates the maximal action of the internal transitions. The inference rules, grouped into three categories, realize the following tasks:

- the rules OBJECT-SEM and MONOTONICITY build, for a given object, its partial transition system according to its methods, places, and behavioral axioms,
- BEH-SEQ, BEH-SIM and BEH-ALT compute all deductible sequential, concurrent and alternative behaviours, while SYNC solves all the synchronizations between the transition systems,
- STAB "eliminates" all invisible or spontaneous events which correspond to internal transitions.

3.6.4 Partial Semantics of an Object

We develop in this section the partial semantics of an object specification, built over an algebra for the algebraic part. This semantics introduce the potential transitions, noted $\rightarrow$, which can be produced by the axioms without considering the synchronization and stabilization process.

Definition 20: *Partial Semantics of an Object*

Let *Spec* be a CO-OPN specification, $A \in Mod(Spec^{\mathsf{A}})$ a model and the Object module $OM = < \Omega, P, X, \Psi >$ of *Spec* where $\Omega^{o} = < \{o\}, M >$. The semantics $PSem_A(OM)$ is the transition system $< Mark_{Spec,A} \times \mathbf{E}_{A,M(Spec),O} \times Mark_{Spec,A} >$, noted $\rightarrow$, and obtained by the rules OBJECT-SEM and MONOTONICITY.

$\forall\ m,m',m'' \in Mark_{Spec,A}$, $e \in \mathbf{E}_{A,M(Spec),O}$, and assignment σ

$$\text{OBJECT-SEM}\quad \frac{(Cond \Rightarrow Event : Pre \rightarrow Post) \in \Psi,\ \exists \sigma : X \rightarrow A,\ \langle A, \sigma \rangle \models Cond}{[\![Pre]\!]^{\sigma}_{[A]} \xrightarrow{[\![Event]\!]^{\sigma}_{\mathbf{E}_{A,M,O}}} [\![Post]\!]^{\sigma}_{[A]}}$$

$$\text{MONOTONICITY}\quad \frac{m \xrightarrow{e} m'}{m + m'' \xrightarrow{e} m' + m''}$$

◊

If we replace the composite event *Event* by a simple transition *t*, then these rules correspond to the semantics of classical algebraic Petri nets [Reisig 91]. In a more procedural approach, we can formulate OBJECT-SEM as follows [Buchs&Guelfi 91][6].

1. An event *Event* defined by the behavioural axiom $Cond \Rightarrow Event : Pre \rightarrow Post$ is *firable* (or *enabled*) if:
   - All the equations of *Cond* are satisfied in *A*: The global conditions are fulfilled for the assignment σ.
   - $\forall\ p$ all the terms of Pre_p are matched[7] by terms of place *p*. This means that the current marking satisfies the preconditions of *Event* both *quantitatively*, i.e. there are enough tokens, and *qualitatively*, because the values of the chosen tokens conform to the algebraic predicates of the precondition, or, in other words, under the assignment σ each term of Pre_p is equal to some token of place *p*.
   - The expression *Event* is satisfied for the assignment σ and its synchronization part is firable. The actual firing of the synchronization part is done only at step 3 below. This decomposition is justified by the possibility of performing recursive method calls as will be described in chapter 6.

2. Before firing *Event* do:
   - Remove the terms of *Pre* from the current marking.

3. When firing *Event* do:
   - Fire the synchronization part of *Event* according to the rules given in 3.6.5.

4. After firing *Event* do:

6. In [Buchs&Guelfi 91] this rule was called EVAL, and was augmented in order to select the object to activate in the specification hierarchy, choice which is now made in the application $Sem^{\sqsubseteq}_{A}(Spec)$ defined below. The ordering of the actions has also been slightly adapted in order to stay coherent with the execution rules to be given later.

7. For the moment we will say that two terms *match* if they are strictly equal, or can be made equal by giving suitable values to their variables. A formal definition will be given in section 4.2.

- Add the terms of *Post* to the current marking.

Notice that it is not specified how a candidate axiom is selected: Among all enabled axioms, any one may be fired at random, provided that the choice respects the additional semantics of CO-OPN (definition 23 below). Likewise, the tokens to be removed from the current marking (step 2 above) may be picked at will as long as they satisfy the conditions of the axiom. This kind of behaviour is said *non-deterministic*, because the evolution of the object depends on an arbitrary choice instead of being determined entirely by its current state.

In an algorithmic approach, a more general difficulty, which includes the problem of the choice of input tokens, resides in the calculation of all possible assignments for rule OBJECT-SEM. The premiss of the rule says: "if there exist an axiom and an assignment such that the global condition is satisfied". In the conclusion of the rule we note that the chosen assignment also intervenes in the actual firing by establishing the interpretation of the precondition, event and postcondition parts. The global condition does however not determine alone the values of all the variables in an axiom. Therefore for the moment we will have to admit that there is some magic which assigns correct values to the remaining variables: This is the purpose of the resolution mechanism. We will see later that in procedural implementations the firing of a rule must follow a prearranged course in order to settle efficiently the value of each variable.

Example 1. *A Simple Algebraic Petri Net: Dijkstra's Philosophers*

This example, taken from [Reisig 91], is Dijkstra's famous dining philosophers problem [Dijkstra 71], expressed as a classical algebraic Petri net, i.e. in a non-modular fashion. We give successively its graphical (figure 20) and corresponding textual (figure 21) specification in the syntax of CO-OPN.

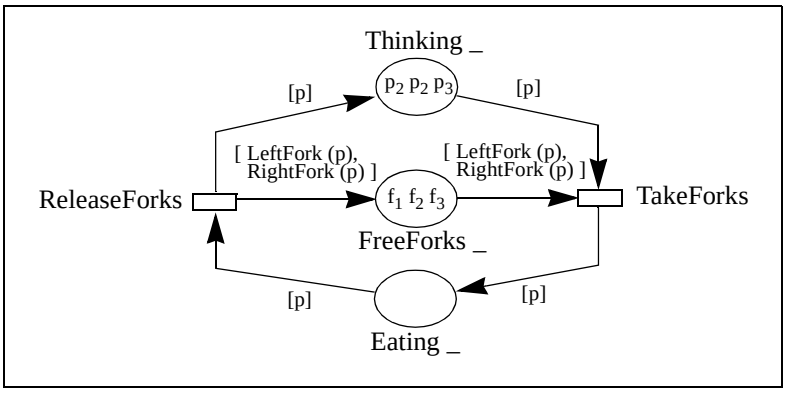


Figure 20. Algebraic Net Formulation of the Dining Philosophers Problem

```
ADT PhilosType;
INTERFACE
  SORTS philos;
  GENERATORS p1,p2,p3 :-> philos;
BODY
  (: No axioms :)
END PhilosType;


ADT ForksType;
INTERFACE
  USE PhilosType;
  SORTS fork;
  GENERATORS f1,f2,f3 :-> fork;
  OPERATIONS
    LeftFork _,
    RightFork _ : philos -> fork;
BODY
  AXIOMS
    LeftFork(p1) = f1; RightFork(p1) = f2;
    LeftFork(p2) = f2; RightFork(p2) = f3;
    LeftFork(p3) = f3; RightFork(p3) = f1;
END ForksType;
```

```
OBJECT Philosophers;
INTERFACE
BODY
  USE PhilosType, ForksType;
  TRANSITIONS TakeForks, ReleaseForks;
  PLACES
    Thinking _, Eating _ : philos;
    FreeForks _ : fork;
  INITIALS
    Thinking p1; Thinking p2; Thinking p3;
    FreeForks f1; FreeForks f2; FreeForks f3;
  AXIOMS
    TakeForks :   Thinking (p),
                  FreeForks (LeftFork(p)),
                  FreeForks (RightFork(p))
              -> Eating (p);
    ReleaseForks :  Eating (p)
                 -> Thinking (p),
                    FreeForks (LeftFork(p)),
                    FreeForks (RightFork(p));
  WHERE
    p : philos;
END Philosophers;
```

Figure 21. CO-OPN Source for the Representation of Figure 20

From the marking of figure 20, we will make the net evolve by firing one of its transitions. We can immediately see that transition `ReleaseForks` is not firable since its input place, `Eating`, is empty. We will then attempt to fire transition `TakeForks` by performing the following steps:

1. Try to match the terms of the preconditions for `TakeForks`:
   - For place `Thinking` the term is `p`, and we can match it with any token of the place: We choose p_1 at random.
   - For place `FreeForks` the terms are `LeftFork(p)` and `RightFork(p)`. Given the value chosen for `p`, these expressions match respectively the tokens `f1` and `f2`.
2. The transition is firable. We must now consume the tokens satisfying the precondition and produce the postconditions.
   - Remove [`p1`] from `Thinking` and [`f1`, `f2`] from `FreeForks`.
   - Add [`p1`] to the place `Eating`.

This firing corresponds to the transition system

$$\begin{pmatrix} \mathit{Thinking}([p_1,p_2,p_3]) \\ \mathit{FreeForks}([f_1,f_2,f_3]) \\ \mathit{Eating}(\emptyset) \end{pmatrix} \xrightarrow{\mathit{TakeForks}} \begin{pmatrix} \mathit{Thinking}([p_2,p_3]) \\ \mathit{FreeForks}([f_3]) \\ \mathit{Eating}([p_1]) \end{pmatrix}$$

The current situation is now depicted in figure 22. Notice that for the evaluation of the preconditions, we have deliberately adopted an ordering of actions which makes the calculation of variable assignments easier. Had we decided to evaluate first the preconditions associated to place `FreeForks`, then it would have been necessary to select randomly a fork `f`, and then to find a solution to the equation $p=LeftFork^{-1}(f)$ in order to ensure that the value of `p` is the same for all preconditions. It is however not possible to compute the inverse of the operation `LeftFork` in the frame of rewrite systems, as will be exposed in chapter 4.

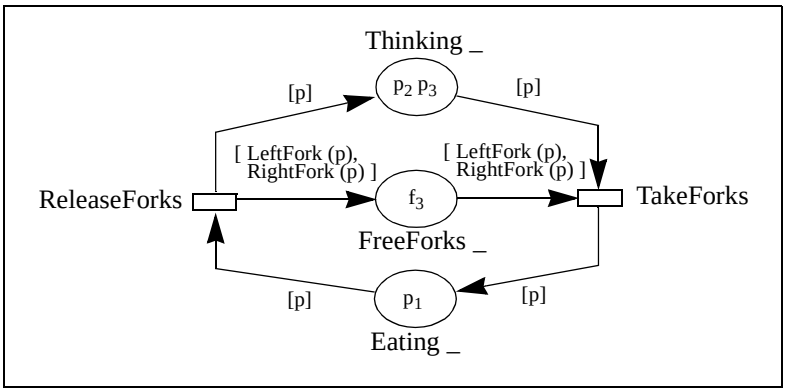


Figure 22. The System of Dining Philosophers After `p1` Having Taken his Forks

3.6.5 Semantics of a CO-OPN Specification

The idea behind the construction of the semantics of a set of Object modules is to consider each partial semantics and to solve the synchronizations requested by the events as well as the stabilization process. This process cannot be performed in random order. This is due to the recursive view of the stabilization process which implies that only already stable modules can be considered when methods are fired. As long as internal transitions can call methods, stabilization can be initiated only on Objects which are themselves built over stable external Object states.

In order to build the semantics we introduce for a given partial order induced by the dependency graph OD_{Spec} a total order $\sqsubset \subseteq OD \times OD$ that will be used to induce the semantics. Given OM_0 the lowest module of the hierarchy and the fact that $OM_i \sqsubset OM_{i+1}$ $(0 \leq i < n)$, we will introduce an all module OM_i $(0 \leq i \leq n)$ semantics from the bottom to the top.

Definition 21: *Closure Application*

Given *Spec* a CO-OPN specification and $A \in Mod(Spec^{\mathsf{A}})$ an algebraic model, *Closure* is the application on transition systems $\rightarrow$ and $*\rightarrow$ induced by the use of the rules BEH-

SEQ, BEH-SIM, BEH-ALT and SYNC.
$\forall\ m,m',m'',m_1,m'_1,m_2,m'_2 \in Mark_{Spec,A},\ \tau,\ e_1,e_2 \in \mathbf{E}_{A,M(Spec),\ O}$

$$\text{BEH-SEQ}\ \frac{m'_1 \uplus m_2,\ m_1 \overset{e_1}{*\!\!\rightarrow} m'_1,\quad m_2 \overset{e_2}{\rightarrow} m'_2}{m_1 \overrightarrow{\cup} m_2 \overset{e_1..e_2}{\longrightarrow} m'_2 \overrightarrow{\cup} m'_1}$$

$$\text{BEH-SIM}\ \frac{m_1 \overset{e_1}{\rightarrow} m'_1,\ m_2 \overset{e_2}{\rightarrow} m'_2}{m_1 + m_2 \overset{e_1 \& e_2}{\longrightarrow} m'_1 + m'_2}$$

$$\text{BEH-ALT}\ \frac{m \overset{e_1}{\rightarrow} m'}{m \overset{e_1\ +\ e_2}{\longrightarrow} m'}$$

$$\text{BEH-ALT}\ \frac{m \overset{e_2}{\rightarrow} m''}{m \overset{e_1\ +\ e_2}{\longrightarrow} m''}$$

$$\text{SYNC}\ \frac{m_1 \overset{e_1\ \textbf{WITH}\ e_2}{\longrightarrow} m'_1,\ m_2 \overset{e_2}{*\!\!\rightarrow} m'_2}{m_1 + m_2 \overset{e_1}{\rightarrow} m'_2 + m'_1}$$

◊

These rules, which form the observable part of a synchronization, will be illustrated and transformed into more procedural semantics in chapter 6. To avoid any confusion, it should be emphasized that they determine rather the behaviour of the called objects than the one of the caller (the emitter of the synchronization).

Definition 22: *Stabilization Application*

Given *Spec* a CO-OPN specification and $A \in Mod(Spec^{\mathsf{A}})$ a model, *PreStab* is the application induced by rules STAB on transition systems $\rightarrow$ and $*\!\!\rightarrow$.

$$\text{STAB}\ \frac{m'_1 \uplus m_2\ ,\ m_1 \overset{e}{*\!\!\rightarrow} m'_1,\ m_2 \overset{\tau}{\rightarrow} m'_2}{m_1 \overrightarrow{\cup} m_2 \overset{e}{*\!\!\rightarrow} m'_2 \overrightarrow{\cup} m'_1}$$

$$\text{STAB}\ \frac{m \overset{e}{\rightarrow} m'}{m \overset{e}{*\!\!\rightarrow} m'}$$

The application *Stab* on transition systems $\rightarrow$ and $*\rightarrow$ build a transition system $*\rightarrow$ by firing all possible transitions until a stable state is reached.

$Stab(\rightarrow \cup *\rightarrow) =$
$\{m *\xrightarrow{e} m' \mid m *\xrightarrow{e} m' \in PreStab(\rightarrow \cup *\rightarrow) \text{ and } \nexists\ m' \xrightarrow{\tau} m'' \in PreStab(\rightarrow \cup *\rightarrow)\}$

◊

The purpose of the second STAB rule is simply to perform the coercion from $\rightarrow$ to $*\rightarrow$ which is needed as first step of every application of *PreStab*.

Notice that the notion of Object is completely absent from the definition of *Stab* and *PreStab*, which suggests that they do not have modular semantics. This will be confirmed later.

Theorem 2: *Stab and Closure are Total Operations*

Given a specification *Spec* with finite stabilization (i.e. there is no infinite succession of internal transitions) and transition systems $\rightarrow$ and $*\rightarrow$, *Stab* and *Closure* are unique and total operations.

Proof: Obvious, due to the structured operational semantics monotonic form of the rules and the finite stabilization. ◊

Since objects are required to be finitely stabilizable, it means that applications which need to model infinite loops may not be directly specified as such. Another caveat is about the presence - intentional or not - of non-determinism within internal transitions, which may induce desastrous performance in the search for stable states during system simulation or actual run-time.

Definition 23: *Semantics of a CO-OPN Specification*

Given a specification *Spec* which includes a set of Objects OM_i, $(0 \leq i \leq m)$ and an algebra $A \in Mod(Spec^{\Sigma})$. The semantics $Sem^{\sqsubseteq}_A(Spec)$ with $(0 \leq k \leq m)$ is defined as:

$$Sem^{\sqsubseteq}_A(\cup_{(0 \leq i \leq k)} OM_i) = \lim_{n \to \infty} (Stab \circ Closure)^n (Sem^{\sqsubseteq}_A(\cup_{(0 \leq i' \leq k\text{-}1)} OM_{i'}) \cup PSem_A(OM_k))$$

$$Sem^{\sqsubseteq}_A(\varnothing) = \varnothing$$

◊

From now on we will note $Sem_A(Spec)$ instead of $Sem^{\sqsubseteq}_A(Spec)$.

Let us precise that the limit of *n* tending towards infinity is needed to cover the special case of recursive method calls: All other situations, where a synchronization goes from an object O_i to an object O_j (where $i > j$), are taken care of by the recursive call to *Sem*.

The semantics expressed by *Sem* is that the behaviour of a set of objects is calculated by starting from the lowest object in the hierarchy and repeatedly adding a new object to the system. We may thus build the graph of all the possible behaviours of a specification. An example is given below.

The mutually recursive nature of the synchronization and stabilization process appears clearly in this definition. Notice also the following points:

- This formulation does not show the potential parallelism of the system, whether it originates from independent activities or from events specified as simultaneous.
- The stabilization order is indifferent, provided that all possible events are fired. Once *Closure* has been applied to the *k* lowest objects of the hierarchy, they must be maximally stabilized. The application *Stab* does however not consider a set of objects as such: They are instead simply viewed as a big transition system. In other words, *Stab* has a global view which allows firing every transition at the very instant it becomes enabled. This infinitely fast and ubiquitous reaction ability is of course impossible to realize in a loosely coupled environment. In chapter 6 we will show how this problem is tackled in wide-area networks, where clever optimizations are needed to avoid exhaustive polling techniques.
- *Sem* lets us reason on subsets of the hierarchy: While working on the lower parts of an object hierarchy, we may safely ignore the expectations of objects which are higher up according to the total order $\sqsubset$. We will also see, in the next example, that this strict obedience to the total order may produce rather counter-intuitive behaviours.

Example 2. *About The Influence of the Total Order*

Let be the following system, where we use only classical “black” tokens instead of algebraic values to simplify the demonstration.

```
OBJECT Ob;                OBJECT Ot;              OBJECT Oa;
INTERFACE                 INTERFACE               INTERFACE
  METHODS b,c;              USE Ob;                 USE Ob;
BODY                      BODY                      METHODS a;
  USE BlackTokens;          TRANSITIONS t;        BODY
  PLACES                    AXIOMS                  AXIOMS
    p _ : blacktoken;         t WITH b;             a WITH c..b;
  AXIOMS                  END Ot;                 END Oa;
    c :-> p token;
    b : p token ->;
END Ob;
```

Figure 23. A Problematic Case for the Total Order

We may number these objects in two ways, both compatible with the object graph OD_{Spec}. The first possibility is to put *Ot* above *Oa*, i.e. the total order gives $Ob \sqsubset Oa \sqsubset Ot$. This results in the following incremental construction of the (infinite) state graph, where the numbers indicate the quantity of tokens in place *p* of object *Ob*:

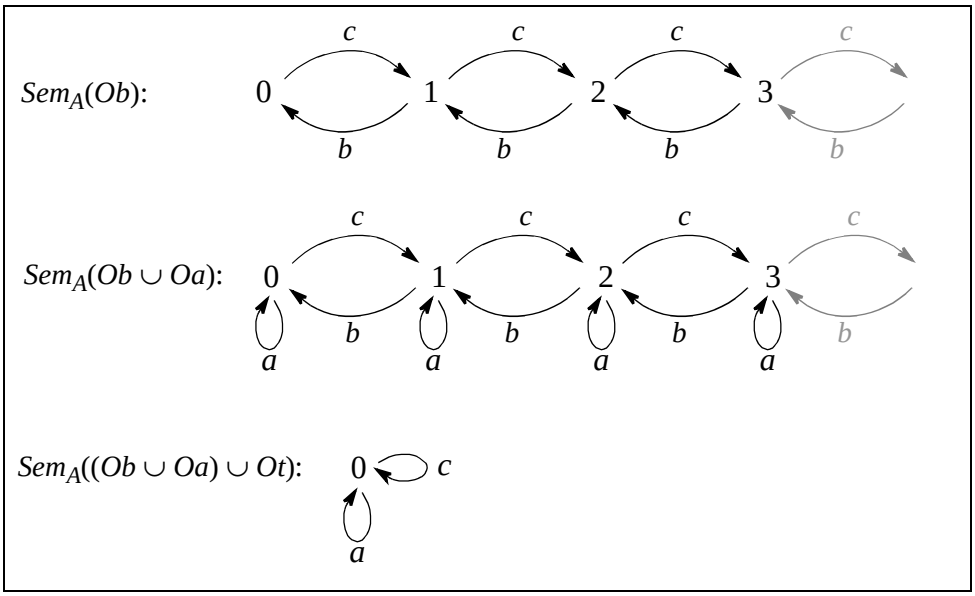


Figure 24. State Graph Construction for $Sem_A((Ob \cup Oa) \cup Ot)$

The resulting shape for the last state graph of figure 24 is due to the fact that transition *t* immediately calls method *b* as soon as it becomes firable. Therefore the system can never offer the method *b* as a service to the outside world once object *Ot* has been incorporated.

The other way of numbering our object modules is $Ob \sqsubset Ot \sqsubset Oa$, which gives the state graphs of figure 25. We can see that, as before, the incorporation of object *Ot* annihilates the effects of a call to method *c* and makes method *b* unavailable. What more is, adding object *Oa* to the system is useless, since the prevalence of transition *t* will always break the atomicity of the sequence *c*..*b* in method *a*.

As a consequence, the establishment of a total order does have some influence on the semantics of a CO-OPN specification: The linearization of the object dependency graph OD_{Spec} constitutes a loss of information(8) and cannot be performed by a simple static observation of the dependencies without encurring the risk of interfering with the developer's intentions. This is still a weak point of the semantics of CO-OPN which requires

8. Interestingly the same kind of problem occurs in the theory of concurrent systems: Keeping a partial ordering of events is semantically richer because it provides a precise description of the dependencies within the real system.

further investigations to be satisfactory, and could result in a hierarchizing with several levels of indexing in order to support different granularities of object clusterings. This approach would still maintain a total order while allowing the developer to indicate implicitly which one of object *Oa* or *Ot* is "closer" to *Ob*.

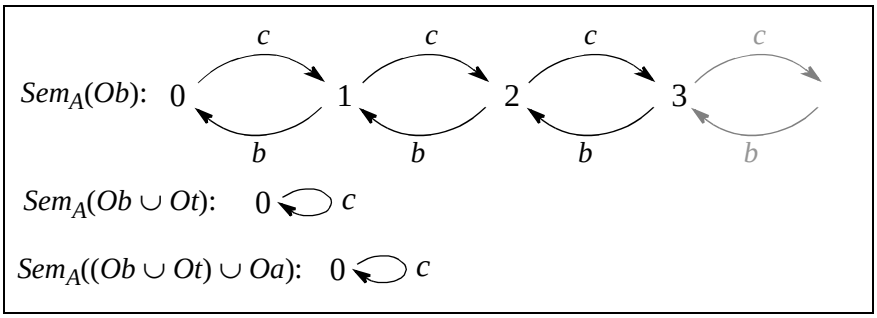


Figure 25. State Graph Construction for $Sem_A((Ob \cup Ot) \cup Oa)$

For the reasons exposed above we will assume in this thesis that the total order is computed and communicated to the implementation by a mechanism which is external and orthogonal to the specification compiler. Our algorithms must then simply conform to the given object numbering.

Theorem 3: *Properties of the Synchronization Operators*

Let *Spec* be a well-formed specification which includes a set of Objects *O* and an algebra $A \in Mod(Spec^{\mathsf{A}})$, and events $e_1, e_2, e_3 \in \mathbf{E}_{A,M(Spec),O}$. The following properties hold for the synchronization operators of CO-OPN:

i) the sequence operator ".." is associative:
$(e_1..e_2)..e_3 = e_1..(e_2..e_3)$
ii) the simultaneity operator "&" is commutative and associative:
$e_1\&e_2 = e_2\&e_1$ and $(e_1\&e_2)\&e_3 = e_1\&(e_2\&e_3)$
iii) the alternative operator "+" is commutative and associative:
$e_1+e_2 = e_2+e_1$ and $(e_1+e_2)+e_3 = e_1+(e_2+e_3)$

Proof: Let be the markings $m_1,m_2,m_3,m_{\tau1},m_{\tau2},m'_1,m'_2,m'_3,m'_{\tau1},m'_{\tau2} \in Mark_{Spec,A}$:

i) In order to prove the associativity of the sequence operator, we have to define events corresponding to the intermediate stabilizations: Let be τ_1 and τ_2 the stabilization (symbolized by a single step) after respective events e_1 and e_2. Their source and target markings are respectively $m_{\tau1},m_{\tau2}$ and $m'_{\tau1},m'_{\tau2}$. Let us first calculate $(e_1..e_2)..e_3$.

- By successive applications of rules MONOTONICITY, STAB and BEH-SEQ, we have:

$$\left(m_1 \overrightarrow{\cup} m_{\tau 1}\right) \overrightarrow{\cup} m_2 \xrightarrow{e_1..e_2} m'_2 \overrightarrow{\cup} \left(m'_{\tau 1} \overrightarrow{\cup} m'_1\right)$$

$$\left(\left(\left(m_1 \overrightarrow{\cup} m_{\tau 1}\right) \overrightarrow{\cup} m_2\right) \overrightarrow{\cup} m_{\tau 2}\right) \overrightarrow{\cup} m_3 \xrightarrow{(e_1..e_2)..e_3} m'_3 \overrightarrow{\cup}\left(m'_{\tau 2} \overrightarrow{\cup}\left(m'_2 \overrightarrow{\cup}\left(m'_{\tau 1} \overrightarrow{\cup} m'_1\right)\right)\right)$$

- Calculating $e_1..(e_2..e_3)$ following the same process, we obtain:

$$\left(m_2 \overrightarrow{\cup} m_{\tau 2}\right) \overrightarrow{\cup} m_3 \xrightarrow{e_2..e_3} m'_3 \overrightarrow{\cup} \left(m'_{\tau 2} \overrightarrow{\cup} m'_2\right)$$

$$\left(m_1 \overrightarrow{\cup} m_{\tau 1}\right) \overrightarrow{\cup}\left(\left(m_2 \overrightarrow{\cup} m_{\tau 2}\right) \overrightarrow{\cup} m_3\right) \xrightarrow{e_1..(e_2..e_3)} \left(m'_3 \overrightarrow{\cup}\left(m'_{\tau 2} \overrightarrow{\cup} m'_2\right)\right) \overrightarrow{\cup}\left(m'_{\tau 1} \overrightarrow{\cup} m'_1\right)$$

The source and target markings of both results are equal, because the marking fusion operator is associative (by theorem 1).

ii) Both properties are obvious, due to the symmetric nature of rule BEH-SIM and the fact that the set union operator also is commutative and associative.

iii) Operator '+' is commutative by definition (rules BEH-ALT). Associativity is proven by showing that all operand groupings lead to the same set of allowed target markings: Let $M_e^n = \{ m_e^1, m_e^2, ..., m_e^n \}$ be the set of all markings resulting from the choice of event e in an expression with the alternative operator. We need this set in order to take into account all other non-deterministic choices made within event e. We define M_{e1}^{n1}, M_{e2}^{n2} and M_{e3}^{n3}, the respective target marking sets of e_1, e_2 and e_3.

- When evaluating $(e_1+e_2)+e_3$ according to the rules BEH-ALT, the choices made in the first and then in the second alternative operator lead to four possible solutions among which two are identical: M_{e1}^{n1}, M_{e3}^{n3}, M_{e2}^{n2} and M_{e3}^{n3}. Therefore $M_{e1}^{n1} \cup M_{e2}^{n2} \cup M_{e3}^{n3}$ is the expression of the set of all allowed solutions.
- If we evaluate $e_1+(e_2+e_3)$, the choices made in the second and then in the first alternative operator also lead to four possible solutions among which two are identical: M_{e1}^{n1}, M_{e1}^{n1}, M_{e2}^{n2} and M_{e3}^{n3}. Here again, the same expression denotes the set of all allowed solutions: $M_{e1}^{n1} \cup M_{e2}^{n2} \cup M_{e3}^{n3}$.

◊

It must be noted that although the sequence and simultaneity operators may be evaluated in any order by virtue of their associativity, it does not change their temporal interpretation. The property of associativity is only applicable to a sequence with the conjoint exploitation

of rule MONOTONICITY, which means that we place ourselves in a deductive frame: It allows a posteriori unification of out-of-order events, i.e. events which happen in an order different from the specification. Likewise, non-concurrent evaluation of events specified as simultaneous does not strictly speaking respect the semantics; the property of associativity simply gives a better understanding of the ressource usage during a synchronization.

Example 3. *Derivation Tree for Recursive Method Calls*

We give a derivation tree showing the necessity of the successive application of *Stab* and *Closure* for the local recursive method calls. Given the following - somewhat artificial - specification:

```
OBJECT Recursive;
INTERFACE
  USE Naturals;
  METHODS
    a _ : natural;
    b;
BODY
  PLACES
    p _ : natural;
  AXIOMS
    b : p(n) -> p(succ(n));
    a(0) : ->;
    a(succ(n)) WITH a(n)..b : ->;
  WHERE
    n : natural;
END Recursive;
```

Figure 26. Object with Recursive Method Calls

We can calculate from the previous inference rules the behavior of `a(succ 0)`(9) by performing successive *Stab* and *Closure* (rules BEH-SEQ and SYNC) applications on the initial behavior, itself computed by the rule OBJECT-SEM, whereas the rule MONOTONICITY introduces additional contextual state when necessary.

9. The evaluation of `succ(0)` is `1` and `succ(1)` is `2` in our algebra of `natural` numbers.

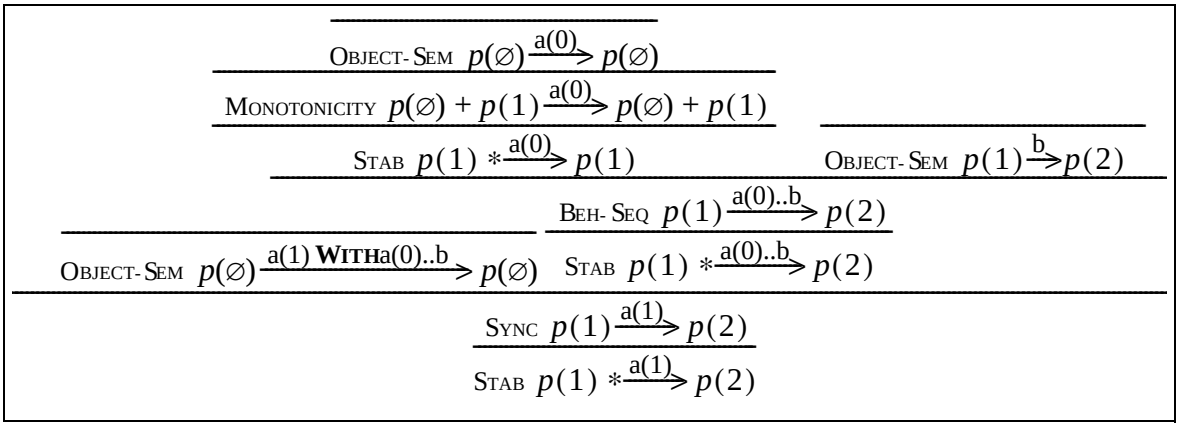


Figure 27. Derivation Tree for a Sequence with Recursive Method Calls

From the derivation tree of figure 27 we can for instance deduce the initial and final object states (contents of place *p*) needed for the call `a(succ 0)` to succeed: Place *p* must initially contain the value 1 and finally the value 2.

Example 4. *Object with Stabilization involving Internal Transitions*

In example 3, there were no internal transitions, which simplified the stabilization process. Now we present a **`Divider`** object (inspired from [Flumet 95]) that needs a stabilization step between method calls in order to perform the actual division (Figure 28).

```
OBJECT Divider;
INTERFACE
  USE
    Naturals;
  METHODS
    PutOpds _ _ : natural natural;    ;; Input of division operands
    RtnRes _ : natural;               ;; Output of division result
BODY
  TRANSITIONS
    Division;
  PLACES
    Opds _ _ : natural;               ;; Contains the couple < dividend,divisor >
    Res _ : natural;                  ;; For the future result
  AXIOMS
    ;; Object discards divisions by zero:
    (y=0)=false => PutOpds(x,y): -> Opds x,y;
    ;; Perform the actual division (operation provided by module Naturals):
    Division: Opds x,y -> Res x/y;
    ;; Object forgets the result after the client has fetched it:
    RtnRes(x): Res x -> ;
  WHERE
    x,y : natural;
END Divider;
```

Figure 28. **`Divider`** Object with Internal Transitions

The **`Divider`** object of figure 28 is sequential in the sense that it can only handle one division at a time, otherwise the answer gets lost. It may be compared with a hardware realization, where the result of each operation must be saved before it is overwritten by subsequent computing: Trying to read the result register later will usually lead to unpredictable behaviour.

Consequently, our **`Divider`** object requires the caller to perform the synchronization `PutOpds(x,y)..RtnRes(r)` in order to get the result *r* which really corresponds to the given operands *x* and *y*. The derivation tree in figure 29 illustrates such a case, with *x*=6, *y*=3 and *r*=2 (the application of rule BEH-SYNC is not shown as we are only interested in the behaviour of the Divider object and not of the client object).

The reason why the sequence guarantees the conformity of the result with the operands is that sequential synchronizations, as any event in CO-OPN, are atomic. In other words, if the synchronization succeeds, it means that the atomicity, among other properties, has been preserved.

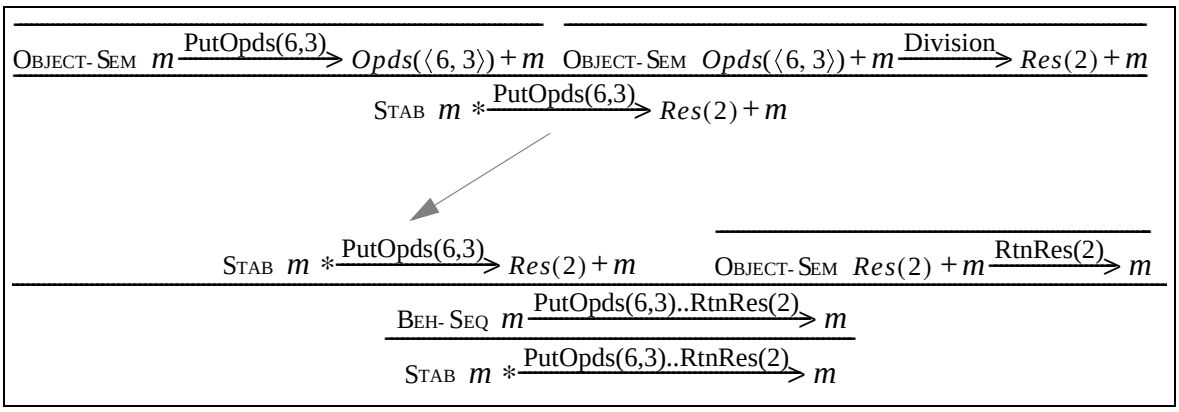


Figure 29. Derivation Tree for a Sequential Synchronization with **`Divider`** Object

The symbolic marking *m* of figure 29 stands for any arbitrary initial state. We notice that *m* is intact at the end of the sequence.

3.6.6 Semantic Discussion About the Diary Example

Since we have introduced operators dealing with the concurrent behavior of objects (operator '`&`'), the transition systems can express evolution steps of concurrent events, i.e.

$$m \xrightarrow{e_1 \& e_2 \& \ldots \& e_n} m'$$

where *m*, *m*' denote two markings and e_1 & e_2 & ... & $e_n \equiv [e_1, e_2, ..., e_n]$ is the multi-set of the (elementary) events e_i (i = 1,...,n).

In the cooperative diary example we have not yet presented the **ADR** Object which manages the low-level accesses to the actual data structures of the diary (see figure 30).

```
OBJECT ADR;
INTERFACE
  USE Booleans, ListEvent, Event;
  METHODS
    Consult _ : listevent;      ;; Give a copy of the diary
    AddEvent _ : event;         ;; Add an entry
    Update _ _ : event, event; ;; Replacement of an entry
    Cancel _ : event;           ;; Erase an entry
BODY
  PLACES
    diary: listevent;           ;; Container for data structure
  INITIAL
    diary [];                   ;; Initialize with empty list
  AXIOMS
    Consult(le) : diary le -> diary le;
    AddEvent(e) : diary le -> diary(e + le);
    ;; The call fails if e1 is not found
    (e1 isin le) = true =>
       Update(e1, e2): diary le -> diary(e2 + (le - e1));
    ;; The call fails if event e is not found
    (e isin le) = true =>
       Cancel e : diary le -> diary(le - e);
  WHERE
    e, e1, e2 : event;
    le : listevent;
END ADR;
```

Figure 30. Specification of Object **ADR**

When serving method `Confirm`, Object **DAL** (Figure 17 on page 47) may want to perform the sequential synchronization stated by axiom `CnfAddEv` with **ADR**:

```
Confirm(AddEvent(e)) WITH AddEvent(e)..Consult(l)
```

Thus, we have, for instance, the following behaviors for the isolated objects after application of rules OBJECT-SEM and BEH-SEQ in the partial semantics:

ADR: $m_2 \xrightarrow{\text{AddEvent(e)}} m'_2 \xrightarrow{\text{Consult(le')}} m''_2$
where $m_2 = diary(le)$, $m'_2 = m''_2 = diary(le')$ and $le' = (e + le)$.

DAL: $m_1 \xrightarrow{\text{Confirm(AddEvent(e)) WITH AddEvent(e)..Consult(le')}} m'_1$
where $m'_1 = m_1 + confirmed(le')$.

If we compose these transition systems using rule SYNC and STAB we obtain:

DAL+ADR: $m_1 + m_2 \xrightarrow{\text{Confirm(AddEvent(e))}} m'_1 + m''_2$

where $m'_1 = m_1 + confirmed(le')$, $m_2 = diary(le)$, $m''_2 = diary(le')$ and $le' = (e + le)$.

Now, if we had two concurrent invocations of method Confirm(AddEvent(e)), then we would use rule BEH-SIM and end up with the following transition system:

$$(m_{1a} + m_{2a}) + (m_{1b} + m_{2b}) \xrightarrow{\substack{\text{Confirm(AddEvent(e}_a\text{))} \\ \text{\& Confirm(AddEvent(e}_b\text{))}}} (m'_{1a} + m''_{2a}) + (m'_{1b} + m''_{2b})$$

where terms indexed by *a* and *b* belong respectively to the first and second arguments of the simultaneity operator. The relations $m_{1a} + m_{1b} = m_1$ and $m_{2a} + m_{2b} = m_2 = diary(le)$ are deduced from rule BEH-SIM, meaning that the source markings m_{1a} and m_{1b} are obtained by splitting the global marking m_1 of **DAL**, and likewise m_{2a} and m_{2b} is obtained from splitting m_2 of **ADR**. The problem is that the marking m_2 cannot be decomposed since it contains a single token, the diary *le*. In other words, the concurrent invocations cannot take place because of the object **ADR**: Only one of them would succeed.

As a rule of thumb, places with non-empty initial markings will usually be dedicated to controlling only the tokens they initialize, and will therefore reduce the potential concurrency of the related methods and transitions. The tokens can be considered as variables protected by a mutual exclusion mechanism. On the other hand, objects like **DAL**, **Network** and **Divider** accept an unlimited amount of concurrent invocations because they are not devoted to the management of a bounded number of tokens.

This concurrent behaviour is a coherent extension of classical Petri nets to the synchronization facility of CO-OPN.

Another lesson learned from this example is that concurrency propagates down the object hierarchy: The simultaneous invocations to method `Confirm` of object **DAL** result in two sequential synchronizations `AddEvent(e)..Consult(l)` being evaluated simultaneously by object **ADR**. It is as if the object had directly received the synchronization

$$\texttt{(AddEvent(e}_a\texttt{)..Consult(l}_a\texttt{))} \;\&\; \texttt{(AddEvent(e}_b\texttt{)..Consult(l}_b\texttt{))}$$

This property is deduced from the resulting markings, and will be discussed further in section 3.8.

3.6.7 Anti-Inheritance of Instability

The semantics *Sem* of a CO-OPN specification corresponds roughly to a classical synchronous method invocation scheme, where the object receives an invocation, tests whether its current state allows treating it, then reacts by doing some internal computing and possibly performing other, so-called *nested*, method calls. It finally returns the result to the caller. This fundamental behaviour may be enriched with synchronization operators, as seen in the previous examples.

There are however situations where it is not sufficient to bear this basic scheme in mind, because of the role of the total order in object hierarchies. Let us recall the details of the semantics *Sem*: It strictly follows the numbering of the hierarchy when deciding what set of objects to stabilize. In other words, after an object O_k has accepted an invocation and performed the first step of the corresponding action ($PSem_A(O_k)$), all sub-objects O_i ($0 \leq i \leq k$-1) will take part in the stabilization, diregarding whether or not they were actually involved in answering a method call initiated in $PSem_A(O_k)$. Consequently, objects may be activated because of state changes due to unrelated or indirectly related events. The following synthetic example illustrates the latter case:

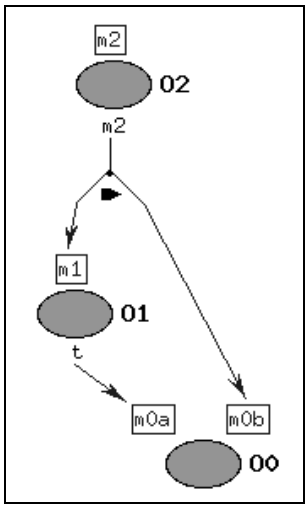


Figure 31. Stability of an Object in Relation with an Enclosing Event

The invocation of m_2 may for instance produce this progression of events (here we only illustrate the phases of *Sem* which produce interesting method calls or replies):

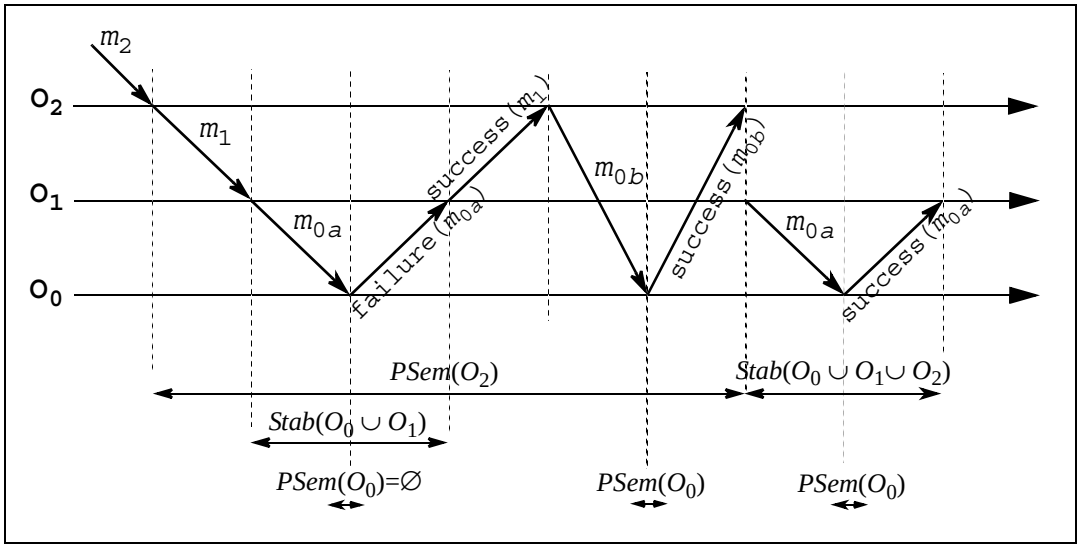


Figure 32. Evaluation of Invocation m_2 (without stabilization messages)

As shown in figure 32, it may happen that invocation m_{0a} only succeeds after object **O0** has been altered by an intermediate method call m_{0b}. This behaviour is fairly unusual, since we would rather expect the call m_{0a} to be considered as belonging to the enclosing call m_1 inside which **O1** first tried to fire it.

A situation where the peculiar stabilization semantics of CO-OPN is frequently exploited is when two objects need to call each other's methods (not imperatively in recursive fashion). It is then necessary to introduce a third object in the specification which is higher in the dependency graph *OD* than the two objects it connects[10]. This auxiliary object will then usually provide no methods, but instead only have transitions which synchronize with the methods of two sub-objects. It then becomes unstable as soon as both of its sub-objects have firable methods. In the collaborative diary example (figure 15), objects **Network** and **DAL** are connected by object **DSA**, the role of which is similar to a *deamon*, in the sense that it is invisible, but automatically wakes up and alerts **DAL** when something is available on the **Network**. At the same time, it allows **DAL** to send data to the **Network**.

To conclude, we may say that the notion of instability is not modular: When an object is unstable, we should consider that all objects above it instantly become themselves unstable, hence the term of *anti-inheritance*. This non-modularity was more evident in [Guelfi 94], where, in order to calculate the behaviour of a set of objects, it was indispensable to collapse the hierarchy into a single equivalent object.

10. We speak here exclusively of CO-OPN v1.5 or earlier, since CO-OPN/2 resolves this limitation by introducing the notion of object reference.

3.6.8 Compositional Properties of CO-OPN Semantics

In this section we will show that it is possible to identify certain structural patterns in the derivation trees which one builds to prove the correctness of a behaviour with respect to the inference rules of CO-OPN. This will serve later for elaborating an algorithmic approach to object execution.

Notice that we do not yet grapple with the problem of finding correct variable assignments, which is implicit in the SOS scheme. Our intention is solely to provide a straight translation of the inference rules into a representation which reveals the compositional properties of the synchronization and stabilization process. This will ease the transition from the currently deductive frame, where the proofs are performed bottom-up, i.e. from the inference rules to the goal to prove[11], to a resolutive method, where the proof trees are built in a top-down manner, i.e. the goal is solved by applying inference rules *backwards* until it is *reduced* to the *empty goal*, or, in other words, until its validity has been demonstrated. The motivation for this conversion is that in the deductive approach the derivations must be guided more or less intuitively in the direction of the goal to verify, otherwise we end up with a completely intractable search space. Hence it is not appropriate for automatic execution (see e.g. [Padawitz 88]). In the case of CO-OPN, the progress is dictated by the application of *Sem*, which tells that *Closure* must be applied before *Stab* on a given set of objects. But within *Closure*, for instance, it is not stipulated exactly which inference rule to choose.

As said before, the operational definition of a search mechanism for selecting suitable axioms and assignments is postponed until chapter 6, where it will be embedded within the synchronization and stabilization algorithms. In fact, we can consider that the resolution of the former problem is nested within the resolution of the latter.

Recall the object with recursive methods of example 3. Its derivation tree for a sequential synchronization was given in Figure 27 on page 74. If we compare this with the derivation tree of the following sample object (Figure 29 on page 75) we notice that the steps required for proving the applicability of rule BEH-SEQ exhibit a similar structure, even though the examples present several differences: The former makes recursive calls, exploits monotonicity and initiates a synchronization, while the latter is passive, but needs internal stabilization for completing the incoming synchronization. This is the proof pattern for rule BEH-SEQ:

11. In a derivation tree, the bottom is in fact constituted by the leaves of the tree, and the top is the root. Therefore the visual impression, which reflects the progress order, may at first seem misleading.

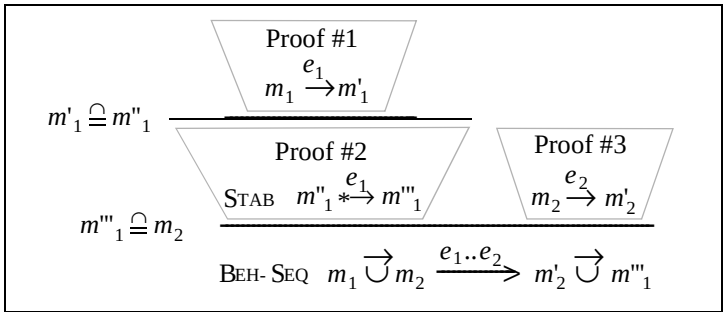


Figure 33. Generic Derivation Tree for rule BEH-SEQ

Proceeding likewise for the other forms of synchronizations, we obtain the figures 34, 35 and 36 hereunder.

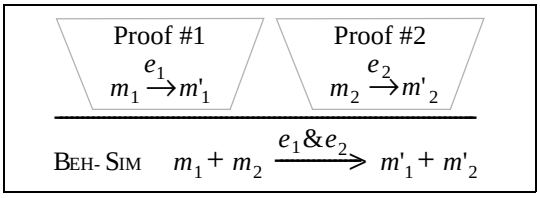


Figure 34. Generic Derivation Tree for rule BEH-SIM

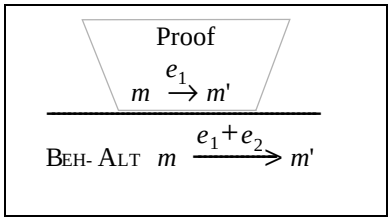


Figure 35. Generic Derivation Tree for rule BEH-ALT

The second BEH-ALT rule is trivially constructed the same way as in figure 35. The SYNC rule is slightly more complex. Notice that proof #1 of figure 36 corresponds to demonstrating that the local state of the caller allows the synchronization to proceed.

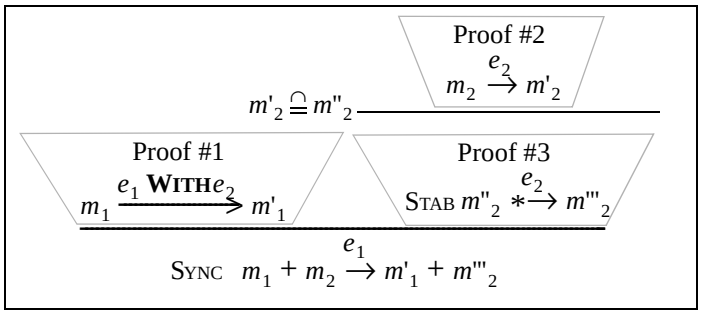


Figure 36. Generic Derivation Tree for rule SYNC

What we have now is a kind of plug-and-play scheme, where the synchronization proofs can be connected directly according to the axioms of the specification. We also notice that this higher abstractness allows the stabilization to be entirely enclosed within the synchronization mechanism.

3.6.9 Summary of the Structured Operational Semantics of CO-OPN

For a well formed specification with finite stabilization and a model *A* of its algebraic part we have given its semantics in terms of transition systems. We have noted that this semantics is well defined, we can add the fact that this semantics is *hierarchical*[12] for the Object part (for the algebraic part we do not detail the necessary modular constraints; they can be found in [Breu 91]). Furthermore, we will see in the section about refinement that the modularity of the semantics leads to separate refinement of Object modules, which constitutes the formal validation of our implementation scheme and incremental prototyping technique applied to CO-OPN.

3.7 Refinement

An important notion which takes part in the software development process from the formal specification up to the actual implementation is the stepwise refinement. Intuitively, a refinement is the substitution of a part of a specification by an another part which is, in general, more concrete, i.e. it has more operational information. Obviously, we expect that such a substitution preserve the semantics of the original specification with respect to the meaning of the term “semantics preservation” we have adopted. Indeed, various levels of

12. In the sense that we can build composite events from more elementary events (We are not saying that the behaviour of an Object Module can be calculated without considering its environment, which would correspond to the usual interpretation of the word *modular*, and which is false for CO-OPN).

semantics preservation may be given. For example, one might be too restrictive and require that both semantics have to be isomorphic. On the other hand, someone else could have a weaker point of view and choose bisimulation or observational equivalence. In other words, saying that one part of a specification is a refinement of another means that it is possible to compare the semantics of these two parts and deduce the semantic preservation.

More formally, but still in an abstract way, we could define as follows what a "good" refinement is. Let *Spec* be a specification and *O*, *O*' be two sets of objects such that $O \subseteq Spec$. *O*' is a refinement of *O* (preserving the signature and interfaces), written $O' \preceq O$, if there exists a relation $\preceq_{Sem}$ such that ('\' being the set difference):

$$Sem(O') \preceq_{Sem} Sem(O) \Rightarrow Sem((Spec \setminus O) \cup O') \preceq_{Sem} Sem(Spec)$$

This definition describes a class of refinements because the semantic comparison criterion is free. This freedom allows one to select the criterion according to ones needs. Thus, one could choose bisimulation, observational equivalence or trace equivalence, for example.

Strong Concurrent Bisimulation

Here we give the definition of strong concurrent bisimulation and we put the previous definition in a concrete form. In other words, we are going to establish that bisimulation stands for our previous refinement definition by the means of theorem 4.

Definition 24: *Strong Concurrent Bisimulation*

Given a model *A* and two concurrent transition systems $TS_{A,1}$ and $TS_{A,2}$. A *bisimulation* between $TS_{A,1}$ and $TS_{A,2}$ is the greatest relation $\mathsf{R} \subseteq Mark(TS_{A,1}) \times Mark(TS_{A,2})$ such that $\forall$ *e* of the form t_1 & t_2 & ... & t_n with the (elementary) events t_i (i =1,...,n):

- if $m_1 \mathsf{R} m_2$ and $m_1 *\xrightarrow{e} m'_1 \in TS_{A,1}$ then there is $m_2 *\xrightarrow{e} m'_2 \in TS_{A,2}$ such that $m'_1 \mathsf{R} m'_2$.
- if $m_2 \mathsf{R} m_1$ and $m_2 *\xrightarrow{e} m'_2 \in TS_{A,2}$ then there is $m_1 *\xrightarrow{e} m'_1 \in TS_{A,1}$ such that $m'_2 \mathsf{R} m'_1$.

We say that $TS_{A,1}$ and $TS_{A,2}$ are *bisimilar* if there exists a non empty relation R between $TS_{A,1}$ and $TS_{A,2}$, and we denote this by $TS_{A,1} \underline{\leftrightarrow}_c TS_{A,2}$. ◊

For a given model *A*, the notation $TS_A(O)$ denotes the concurrent system associated to the semantics of the set of objects *O*. Note that in the following theorem we consider a unique algebraic specification and transition system built up from the same algebraic model.

Theorem 4.

Let *Spec* be a well formed CO-OPN specification and *O* and *O*' be two well-formed sets of objects such that $O \subseteq Spec$ and $O' \subseteq Spec$.

$$TS_A(O')\,|_O \underline{\leftrightarrow}_c TS_A(O) \Rightarrow TS_A((Spec \setminus O) \cup O')\,|_{Spec} \underline{\leftrightarrow}_c TS_A(Spec).$$

The notation $TS_A(O')\,|_O$ means that the concurrent transition system associated to the semantics of *O*' is limited to the elements of *O*.

Proof: An obvious extension of the proof of theorem "replacement of objects" presented in [Buchs&Guelfi 91]. ◊

The previous results are useful for establishing the validity of our automatic implementation process, since it must provide compatibility with respect to $\underline{\leftrightarrow}_c$.

3.8 A Characterization of CO-OPN Events

In this section we will build upon the remarks made in the examples of the previous section in order to characterize CO-OPN object activity in terms of notions such as concurrency, atomicity and strong synchrony. This discussion will help us understanding CO-OPN better and developing efficiently executable semantics.

As a preliminary remark, let us recall that the states of CO-OPN objects should be considered as *almost* private: While the data is completely encapsulated, the notion of instability is automatically and instantly anti-inherited, i.e. inherited upwards, in the object hierarchy. In this report we will however do our best to express the semantics in a modular way, keeping in mind that the objective is to produce distributed implementations of CO-OPN specifications. To simplify, we consider for the moment that each object module executes on a different node and can only communicate with other objects by means of method calls.

3.8.1 Method Calls Viewed as Rendez-Vous

As already mentioned, the method call mechanism of CO-OPN is synchronous, in the sense that the caller is blocked until reception of a reply. Moreover, the callee decides, by means of the global conditions and preconditions associated to the called method, whether it is ready to accept the request. Therefore we may consider the method call as a rendez-vous, where the global conditions and preconditions correspond to the notion of *guards* of other formalisms (e.g. CSP [Hoare 78]) and languages (e.g. Ada [Ada 83]).

As opposed to other known systems, CO-OPN not only allows data to be exchanged in both directions through the parameters of the called method, but also provides operators which state in a very high-level fashion how an object is to synchronize with several other objects within the same rendez-vous.

3.8.2 About Instantaneity and Strong Synchrony

According to [Buchs&Guelfi 91], CO-OPN might be assimilated to the class of *strong synchronous languages*, like Esterel [Berry&Gonthier 88] and Statecharts [Harel 87]. This means that every operation, or change of state, be it internal or external, is viewed as being instantaneous: There is a global discrete time scale. As a consequence, nothing happens between two consecutive instants. Everything must happen as if the processors running the program were infinitely fast[13]. This is the *synchrony hypothesis*. But the analogy stops here, because the finalities of the respective formalisms are quite different.

In the case of languages like Esterel, the property of instantaneity was designed as a means of mastering and reasoning about *reactive* real-time systems. Reactive systems are computer systems which react continuously to their environment, at a speed determined by the latter [Harel&Pnueli 85]. This class of systems contrasts, on one hand with transformational systems (i.e. programs, the inputs of which are available at the beginning of their execution, and which deliver their output when terminating), and on the other hand with interactive systems (which react continuously to their environment, but at their own speed, for instance like classical operating systems). In order to guarantee a predictable response rate, synchronous languages have deliberately restricted themselves to programs that can be compiled into finite deterministic automatons [Caspi&Girault 95]. This is clearly not the case of CO-OPN.

In CO-OPN, internal transitions being instantaneous is due to the particular interpretation of the Petri nets as a synchronous Petri nets model. Moreover, concerning the synchronization scheme of CO-OPN, the objective is to have a convenient way of hiding away the behaviour of external components. This results in a manifest enforcement of the hierarchical relationship between object modules and constitutes a powerful abstraction mechanism which is supported by a solid formal basis [Guelfi 94]. Therefore the notion of instantaneity in CO-OPN is to be considered more as a modelling facility than as constraint on the implementation.

The point of view that all events are instantaneous may be taken as a transposition of atomicity into the temporal dimension. Informally, we could add that two events which are both instantaneous and truly concurrent will appear as being simultaneous, since they will start

13. This is why infinite loops do not fit well within the model of CO-OPN, hence the hypothesis that all stabilizations eventually terminate.

and end pairwise at the same moment. This justifies the appellation of *simultaneity operator* for the concurrent composition primitive of CO-OPN.

3.8.3 Ordering of Events in CO-OPN

Let us recall the signification of rule BEH-SEQ: In a sequential synchronization $e_1..e_2$, the state resulting from e_1 is the same as the source state for e_2 (for what concerns the common places). In other words, no other event e_3 can happen during the sequence and modify this intermediate state. The sequence $e_1..e_2$ forms an indivisible event enclosing both the observable (e_1 and e_2) and invisible (the respective stabilizations) events. This behaviour may be explained by the way CO-OPN considers time: Due to the requirement of instantaneity, each event will happen either strictly before, or after, or yet simultaneously with any other event.

Process algebras (e.g. CCS [Milner 80], CSP [Hoare 78]) reduce the behaviour of concurrent systems to a set of arbitrary sequences by justifying that a sequential observer cannot distinguish the concurrent occurrences of two actions from their occurrence as *interleavings*. On the other hand, Petri nets permit a partial ordering of events because they preserve the causality relationships inside the system. In other words, net semantics allows independent actions to really take place at the same time, not excluding each other, hence the term of *true concurrency.*

In truly concurrent systems independent activities are allowed to partly overlap. *Strong concurrency*[14] is a restriction which says that concurrent activities start exactly at the same moment. Relaxing this constraint results in *weak concurrency.* Finally, if we add the requirement of simultaneity, as in CO-OPN, then concurrent events will also have to end precisely at the same time (because their treatment takes no time). The following figure illustrates these distinctions by means of local activities within two objects O_0 and O_1. In [Fromentin&Raynal 94] different forms of concurrency are expressed instead by relationships between local states, in order to generalize the assumption of instantaneity of events. Figure 37 can also be interpreted that way.

14. The notion of *strong concurrency* appears in the field of Petri nets for the first time in [Reisig 86]. It is defined as an invariant (observer independent) property of concurrent state/transition sequences which appears when two sub-parts of a net show special forms of dependency. The general concurrency relationship is thus the union of strong concurrency and the classical form of concurrency, which is based on independency.

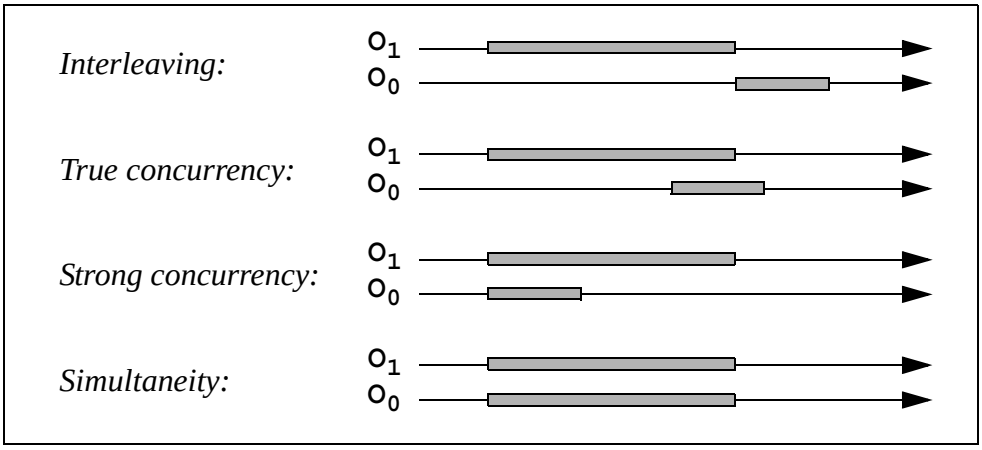


Figure 37. Different Forms of Concurrency

As seen in sub-section 3.6.6, concurrency in CO-OPN is propagated down the hierarchy, possibly resulting in concurrent threads of execution within *shared objects*[15], provided that the required number and values of input tokens are present. An interesting property, deduced from rule BEH-SIM, and intuitively enforced by the instantaneity of events, is that these threads can not interact. It is as if every simultaneously shared object was split into independent sub-objects, ignoring each other, among which the initially available tokens were distributed. The sub-objects merge together when they have all served their respective method calls. At last the result of this fusion is stabilized.

The **`Divider`** object (Figure 28 on page 74) is an example which exploits the atomicity and instantaneity of rule BEH-SEQ, since it requires its clients to call the methods `PutOpds` and `RtnRes` in strict sequence in order to avoid unpredictable behaviours.

The semantics of strong concurrency eliminates all forms of interleaving-based indeterminism. It unambiguously determines the interpretation of multiple synchronization requests reaching a shared object, both when they are parts of the same root event and when they are independent, i.e. of different origins. We will see in chapter 6 that they may lead to different implementation strategies.

3.8.4 Kinds of Non-Determinism in CO-OPN

We can distinguish within the CO-OPN formalism several kinds of non-determinism, resulting from sources of different nature. We can mention:

15. A *shared object* is an object which receives multiple method invocations. We say that it is *shared sequentially* if two of those invocations are synchronized by the sequential operator, and likewise for *simultaneous* and *alternative sharing*.

1. Non-determinism resulting from the choice of input tokens in transitions and method calls. As told in section 3.6.4, this selection is performed randomly, in conformity with the semantics of Petri nets. When dealing with classical 'black' tokens, there is no such thing as a bad choice, since all tokens are equivalent, on the opposite of coloured and algebraic Petri nets. Talking about the operational mechanism needed for CO-OPN, a wrong decision can be corrected very easily in the case of simple events, but if there is a synchronization, then copies of the token values may have been passed on to other objects, and consequently there will be more work to undo in order to correct the situation. This is a form of *don't know non-determinism*, also called *search non-determinism*, since it requires the ability to *backtrack*, i.e. to step back to the choice point, in order to try another token combination.

The three next kinds of non-determinism are all related to the choice of the next axiom to evaluate, but clearly correspond to different semantic problems:

2. Non-determinism resulting from the declarative form of the language, as in Prolog. Since there is no if-then-else construct, each alternative is described by a different axiom. The execution mechanism will then have to try all alternatives until it finds one which evaluates to 'true'. Furthermore, since these alternatives are not necessarily mutually exclusive, there must be an arbitrary choice between the valid ones. This is also a form of don't know non-determinism.
3. Non-determinism from the unspecified ordering of steps in the stabilization process. This is the same kind of non-determinism as in classical Petri nets and emanates from structural conflicts and concurrency of independent events. This is sometimes called *indeterminism.*
4. Non-determinism induced by the presence of inter-object concurrency. This is why objects must be ready to receive any method invocation at any moment and therefore specify completely, by means of a guard mechanism, its willingness to accept a synchronization request. This is also a form of *indeterminism.*

The second and fourth forms of non-determinism are hard to discriminate, since the syntax of CO-OPN does not promote any notion of guard. The global conditions and preconditions of an axiom describe collectively and undistinguishably the guard, which allows to select a rendez-vous, as well as the conditions local to the chosen rendez-vous.

The concept of guard was initially introduced in [Dijkstra 75] for parallel algorithmic languages. The family of so-called *commited-choice* languages (e.g. *Parlog* [Clark 84], *Concurrent Prolog* [Shapiro 83], *Guarded Horn Clauses* [Ueda 85]) adopted this mechanism as a means of reducing the implementation cost of parallel Prolog programs. This was realized by eliminating don't know non-determinism in favour of *don't care non-determinism*: Once a computation branch has been chosen, it is no longer possible to return to the

choice point to explorate other paths. Concretely this is accomplished by incorporating the *commit operator*, the role of which is to separate the guard from the body of the clause and to tell where the speculative parallel evaluation of a goal can end and the trusted sequential execution begin. More importantly, these languages sacrifice the simplicity and completeness of sequential Prolog by abandoning the support of backtracking.

Some versions of distributed Prolog, e.g. CS-Prolog [Ferrenczi&Futo 92], present both kinds of non-determinism, but with a syntax which allows the separation of the guard from the clause body, and with the restriction that the parameters of the goal must be beforehand instantiated by the caller in order to ensure independency between concurrent processes.

We may thus say that CO-OPN is ‘more declarative’ than these parallel and distributed Prolog dialects(16), which seems rather reasonable for a specification language since its objective is to state only *what* must be done, and not *how* to do it. This concern is expressed by the famous equation *program = logic + control*.

3.9 Epilogue

We have now achieved the presentation of the aspects of CO-OPN which are relevant to the prototyping process. These aspects are essentially of semantic nature, as our desire is on the one hand to detail the knowledge which is necessary for applying incremental prototyping to CO-OPN specifications (chapter 5 and 7), and on the other hand to prepare for the transition to the algorithmic description of the distributed run-time support (chapter 6). The chapter 4, which follows, describes the operational semantics that we have adopted for prototyping algebraic abstract data types.

16. We have deliberately chosen to talk only about Prolog dialects with explicit concurrency. Other forms exist, which focus on exploiting the parallelism which is intrinsic to the declarative nature of the language: Or-parallelism, and-parallelism and stream-parallelism. These aspects, although very useful, are not taken into account here since our objective is not to provide efficient implementations to programs which are primarily sequential, but rather to study how search non-determinism is integrated into inherently concurrent systems.

Chapter 4

Operational Semantics of AADTs

The purpose of this chapter is to show how algebraic abstract data types (AADTs) are interpreted in the frame of our prototyping scheme. To this end, we give the operational semantics, the compilation algorithm, and the (intermediate) target language semantics. Most of this work results from slight modifications of the paper [Schnoebelen 88], changes which are necessary to take into account the requirements of mixed prototyping. These are essentially dictated by the desire to make the generated code more readable, as well as to exploit extensively the dynamic binding capabilities of the object-oriented paradigm.

The contents of this chapter are beneficial for understanding the compilation of Adt modules taken alone, but also for the use of AADTs which is made in algebraic Petri nets. In particular, the filtering algorithm which is employed for implementing the operations of an AADT can be reused, with minor adaptations, to realize the pattern-matching needed for selecting appropriate tokens in the places of an algebraic Petri net.

The actual code generation in a target object-oriented programming language is described in the next chapter, along with the possibilities for prototyping AADTs.

4.1 Introductory Example

Before studying the compilation algorithm, it may be interesting to see the kind of code it produces. In the interest of clarity, we will show the final result in Ada95, instead of the abstract representation used by the algorithm.

For instance, let be a simple Adt module **`Stack`** of natural numbers, like in figure 38:

```
ADT Stack;
INTERFACE
  USE Naturals;
  SORT stack;
  GENERATORS
    empty : -> stack;
    push _ on _ : natural stack -> stack;
  OPERATIONS
    top : stack -> nat;
    pop : stack -> stack;
BODY
  AXIOMS
    top(push x on s) = x;
    pop(push x on s) = s;
  WHERE
    x : natural;
    s : stack;
END Stack;
```

Figure 38. Specification of a **`Stack`** of natural numbers

Given the specification of figure 38, the code generated for operation `top` will look like the function in figure 39 below. We precise that mixed prototyping, when applied to AADTs, requires the generator inverse functions to be defined, either automatically or manually, according to which internal representation is chosen (see next chapter). In the **`Stack`** case, the inverse of the generator `push_on` is named `push_on_Inv` and returns a structure containing the original arguments of the call which created the given stack element. For instance, if the stack is instantiated by a call to `push_on(x,s)`, then `push_on_Inv` returns the couple < `x`, `s` > of type `push_on_Arg` inside which the elements are accessed through the respective names `Nat1` and `Stack1`.

```
1   function top (P: in Abstract_Stack) return Natural is
2   begin
3     -- axiom: top(push x on s) = x
4     if Generator(Stack(P))=push_on then
5       declare
6           inverse: push_on_Arg := push_on_Inv(Stack(P));
7       begin
8           return inverse.Nat1.all;
9       end;
10    else -- Error: P = empty
11      raise constraint_error;
12    end if;
13  end top;
```

Figure 39. Ada95 Code Generated for Operation `top`

The given compilation algorithm produces the code corresponding to the lines 4 to 12 of figure 39. The rest is mere packaging, and is the role of the target language-dependent code generator. Let us now proceed with the formal definitions.

4.2 Rewrite Systems

In this section we will briefly remind the notion of *rewrite system*. For deeper explanations, we refer to [Dershowitz&Jouannaud 90]. We also recall that AADTs [Ehrig&Mahr 85] were already defined in section 3.4.1.

Most tools aimed at evaluation of AADTs are in fact theorem provers which can be realized, in the case of positive conditional axioms, using a Prolog interpreter. This is necessary because the expressive power of equational specifications is so high that it requires a very compute-intensive algorithm [Knuth&Bendix 70] to implement them directly.

Instead we will consider the specifications as *conditional term rewriting systems*, which means that the equations are oriented. This approach is most appropriate in the context of prototyping since it allows more efficient executions, provided that we restrict the specifier's freedom as shown below (definition 28, in addition to the prohibition of equations between constructors).

Within the field of term rewriting systems, there exist two execution strategies. The first one consists in an *interpreted mode*, i.e. the rewrite process is performed literally by manipulating syntactical substitutions "as by hand". The other form of execution resides in evaluating the specified functions simply by calling a corresponding function formerly generated in a compiled programming language [Kaplan 87]. It is the latter solution which is exploited in the frame of mixed prototyping, since it provides executions which are several orders of magnitude faster, and also because this approach directly furnishes the required initial abstract implementation.

Definition 25: *Positions in a Term*

Let $\Sigma = < S, F >$ be a complete signature and X be an S-sorted variable set. Each term $t \in T_{\Sigma,X}$ is viewed as a tree of symbols in Σ and X. A *position* in a term t is a sequence of natural numbers, and the *set of positions* of t is written $Pos(t)$ and is defined inductively by:

- ε is the root position of the term t;
- if p is a position of the term t and t has n subterms, then $p.i$, $i \in [1..n]$, denotes the position of the i^{th} subterm.

The symbol occurring at position p in t is written $t[p]$. The subterm occurring at position p in t is written $t|_p$. ◊

For example, if $t = f(g(x,a),h(x))$, we have $Pos(t)=\{\varepsilon, 1, 1.1, 1.2, 2, 2.1\}$, and the expression $t[2]$ denotes the symbol h, while $t|_2$ is the term $h(x)$.

Definition 26: *Ordering of Positions*

We define « as the smallest transitive relation such that $\forall\, p \in \mathbb{N}^*, \forall\, i \in \mathbb{N}, p \ll p.i$. This notion also exists for tuples of terms: $p \in Pos(t_i)$ iff $i.p \in Pos(<t_1,..., t_n>)$. ◊

Definition 27: *Substitution*

Let $\Sigma = < S, F >$ be a complete signature and X be an S-sorted variable set. A *substitution* is an application $\sigma : X \rightarrow T_{\Sigma,X}$. The *domain* of σ is the set of variables that are actually modified by σ: $Dom(\sigma) = \{\, x \in X \mid \sigma x \neq x \,\}$. The substitution of $x \in X$ by $t \in T_{\Sigma,X}$ is noted t/x. ◊

Definition 28: *Conditional Rewrite Rules*

An n-ary operation $op \in OP$ is defined by a list of n-ary *conditional rewrite rules* $\pi \Rightarrow \lambda \rightarrow \rho$, where:

1. λ is a (n-ary) linear tuple of T_{C,X^n};
2. $\pi, \rho \in T_{F,X}$;
3. π has the predefined sort *meta-boolean*;
4. $Vars(\rho) \cup Vars(\pi) \subseteq Vars(\lambda)$.

π is called the condition, which by default is equal to “true”, and may be composed of conjunctions of sub-conditions which are in turn equalities of terms. We use the name “meta-boolean” because it is predefined and not to be confused with potential user-defined boolean sorts.

Evaluating a term of T_F means rewriting it into a term of T_C. If this operation is possible through a finite sequence of applications of rewrite rules, then the rewrite system is said *finitely terminating*. If applying different sequences of rewrite rules to the same term always give the same resulting term, then the system is *confluent*. If a rewrite system is finitely terminating and confluent, which we will assume from now on, then we can prove that every term admits a unique *normal form*, i.e. a term on which no more rewrite rules can be applied.

A term t_1 is said to *match* a term t_0 if there exists a substitution σ such that $t_1 = \sigma t_0$, and this is written $t_0 \preccurlyeq t_1$. We write σ_g the set of all *grounding substitutions*, i.e. substitutions from $T_{F,X}$ into T_F. Given a term $t \in T_{F,X}$ we write $G(t)$ the set of its ground instances, that is $\{\sigma t \mid \sigma \in \sigma_g \}$. ◊

4.3 Semantics of the Source Language

We assume that the rewrite systems to compile are confluent and finitely terminating. Given the above definition of functions, the left-hand side of their equations are required to be linear (rule 1). This is a restriction due to the chosen compilation algorithm [Schnoebelen 88]. Several occurrences of the same variable must be manually transformed into an additional condition stating equality of the corresponding arguments. Thus,

```
c1 & ... & cn => f(x,x,y,z) = v(x,y,z);
```

is the same as:

```
c1 & ... & cn & (x = x') => f(x,x',y,z) = v(x,y,z);
```

This restriction, although a bit annoying, does not reduce the expressivity of the accepted source language. It may however require the definition of additional operations for *observing* the data types, in order to compensate for the prohibition of pattern-matching. For instance, the operation `remove_duplicates` scans a list built by calls to the generator `[_|_]` (where the first argument is the head and the second is the tail of the list) and removes one element of each pair of consecutive duplicates. One of the axioms could be written like this:

```
remove_duplicates([x | [x | tail]]) = [ x | remove_duplicates(tail)];
```

The implicit condition stated by using twice the variable *x* in the left-hand side must be made explicit by a call to the equality operator:

```
x=y => remove_duplicates([ x | [ y | tail]]) = [ x | remove_duplicates(tail)];
```

The evaluation is performed according to the very classical *strict* and *innermost* strategy, which corresponds to the *call-by-value* parameter passing mechanism of most modern imperative programming languages.

It is not specified whether the terms are evaluated from the left or from the right, since it does not alter the final result in confluent rewrite systems(1).

We give below the semantics of the source language. $Rules(op) = (\pi_i \Rightarrow \lambda_i \rightarrow \rho_i)_{i=1...m}$ designates the list of rewrite rules defining the operation *op*.

1. The actual computation order is determined by the target language rules or compilers.

Definition 29: *Application 'Rewr'*

For all $c \in C$, $f \in F$, and $op \in OP$, *Rewr*: $T_F \rightarrow T_C \cup \{\texttt{error}\}$ is defined by:

(S1) $$\frac{Rewr[t_i] = \texttt{error}}{Rewr[f(t_1,...,t_n)] = \texttt{error}}$$

(S2) $$\frac{\forall\ i = 1...n,\ Rewr[t_i] \neq \texttt{error}}{Rewr[c(t_1,...,t_n)] = c(Rewr[t_1],...,Rewr[t_n])}$$

(S3) $$\frac{\forall\ i = 1...n,\ Rewr[t_i] \neq \texttt{error}}{Rewr[op(t_1,...,t_n)] = Apply[Rules(op)]\ op(Rewr[t_1],...,Rewr[t_n])}$$

Figure 40. Semantics of *Rewr*

◊

Definition 30: *Application 'Apply'*

For all $op \in OP$ and $\sigma \in \mathbf{\sigma}_g$, *Apply*[*R*]: $T_C^n \rightarrow T_C \cup \{\texttt{error}\}$ is defined as:

(S4) $Apply[\emptyset]\ op(t_1,...,t_n) = \texttt{error}$

(S5) $$\frac{\lambda_1 \not\preccurlyeq\ op(t_1,...,t_n)}{Apply[(\lambda_i \rightarrow \rho_i)_{i=1...m}]\ op(t_1,...,t_n) = Apply[(\lambda_i \rightarrow \rho_i)_{i=2...m}]\ op(t_1,...,t_n)}$$

(S6) $$\frac{\sigma\lambda_1 = op(t_1,...,t_n) \quad Rewr[\sigma\pi_1] = \texttt{true}}{Apply[(\pi_i => \lambda_i \rightarrow \rho_i)_{i=1...m}]\ op(t_1,...,t_n) = Rewr[\sigma\rho_1]}$$

(S7) $$\frac{\sigma\lambda_1 = op(t_1,...,t_n) \quad Rewr[\sigma\pi_1] = \texttt{false}}{Apply[(\pi_i => \lambda_i \rightarrow \rho_i)_{i=1...m}]op(t_1,...,t_n) = Apply[(\pi_i => \lambda_i \rightarrow \rho_i)_{i=2...m}]op(t_1,...,t_n)}$$

(S8) $$\frac{\sigma\lambda_1 = op(t_1,...,t_n) \quad Rewr[\sigma\pi_1] = \texttt{error}}{Apply[(\pi_i => \lambda_i \rightarrow \rho_i)_{i=1...m}]\ op(t_1,...,t_n) = \texttt{error}}$$

Figure 41. Semantics of *Apply*

◊

Observe that the definitions of *Rewr* and *Apply* are mutually recursive.

When evaluating a term, it may happen, if the rewrite system does not respect *sufficient-completeness*, that none of the equations cover the particular case, leading to the rule S4. The result is then an error, or in terms of implementation, an exception is raised.

From rule S5, we deduce that when several rules of the specficition may apply, the first one - in the source text - is always selected. This enforces the procedural side of specification. If this effect is not desired, the user must take care that none of the equations overlap, or have the compiler issue appropriate warnings.

A final issue concerning the source language is that we do not allow equations between generators. There are two reasons for this. First, rewrite systems with such equations need to undergo some transformations to be compilable, and in mixed prototyping we do not want to alter the structure of the source specifications: This decision was taken to help the developer stay familiar with his model, the structure - i.e. the operation names - being the sole hook he has to orient himself inside the generated code. Second, as shown in [Choppy et al 89], contructors with equations need very special inverses, programmed as iterators, which breaks the orthogonality of the development methodology. This problem is related to the difficulty of cleanly implementing search non-determinism in procedural languages. This subject will be deepened in chapter 6, in the frame of CO-OPN objects, for which we unfortunately cannot avoid supporting this kind of non-determinism.

4.4 Semantics of the Target Language

In this section we describe the semantics expected from the target language *TL*. It must be noted that this target language is only the result of the compilation algorithm. As thus, it used exclusively as an internal representation before the actual code generation in a target programming language. Interestingly, by recurring to this abstract intermediate representation, the compilation algorithm does not have to take into account the specificities of the final implementation language: For instance, there is nothing at this stage which shows that we aim object-oriented programming languages.

Concerning the correctness of mixed prototyping with respect to the semantics of the source specifications, we refer to [Choppy&Kaplan 90]. Although our implementations make heavy use of dynamic binding, we will not bother to define this property formally, because the selection of the pertinent functions for a given implementation is entirely determined at program startup and does not change during the whole execution. Hence we reduce the correctness proof to the case where function calls are resolved statically.

In order to adapt the code generation to OOMP we slightly change the syntax of the testors by making them as implementation-independent as possible through the utilization of polymorphism. The purpose of a testor is to support pattern-matching: It verifies the tag of a value, i.e. finds out which generator it was created by. In the field of mixed prototyping we want to perform this test while staying free from implementation details, since this tag information will usually not be directly available in concrete classes. This is achieved by having the derived classes define a function, call it "`Generator`", which returns an enumerated type corresponding to the different generators of the given AADT. The correct version of the function is executed through dynamic binding. Take for instance the following axioms for calculating the number of elements in a stack:

```
card(empty)=0;
card(push x on s)=1+card(s);
```

In a classical implementation of AADTs, the code generated for the first axiom would be:

```
if object.tag=empty then return 0;
```

This of course obliges all internal representations to have an attribute called 'tag'. The symbol `empty` is of an enumerated type. The abstract code given in [Choppy&Kaplan 90] for mixed prototyping is:

```
if object=empty then return 0;
```

Here the symbol `empty` denotes a constant object. It is the developer's burden to export such an object from the concrete code and to verify that the equality operator corresponds to the semantics of his own data type definition. A safer solution is where there is for each generator one testor function initially declared as abstract. The developer is then obliged to provide a definition for each:

```
if empty_tag(object) then return 0;
```

Finally, our definitive proposal is the following, where we limit to one the number of testor functions to be redefined. Here again `empty` belongs to an enumerated type:

```
if Generator(object)=empty then return 0;
```

The portable testor function has in fact the same role as the *generator inverse* w.r.t. selector functions in the jargon of AADTs.

Definition 31: *Target Language TL*

The *target language TL* has respectively Expressions *E*, Meta-Boolean expressions *B* and Terms *Te*:

TL = { E, B, Te }, where
E ::= *Te* | `if` *B* `then` *E1* `else` *E2* | `no_match`

$$B ::= \texttt{Generator}\,(Te) = c \mid Te_1 = Te_2$$
$$Te ::= x \mid c^{-1}(Te) \mid op(Te_1,...,Te_n)$$

Symbol x denotes any variable, c any constructor, op any n-ary function and p any position. *Generator* is a testor function which calculates the normal form of its argument and returns the most external symbol (at position ε) of the resulting term. The generator inverse is written c^{-1} and returns a tuple - the list of arguments of the given generator - whose member selection is noted "|". ◊

The semantics is given by figures 42 and 43: This is in fact the output of the compilation algorithm given in section 4.5. $Eval_\gamma$ evaluates an expression E in an environment γ where γ is a grounding substitution that assigns values to the variables of X. The empty list is noted ε.

Definition 32: *Application $Eval_\gamma$ for Domain TL_E*

For all $x \in X$, $f \in F$, $c \in C$, $p \in \mathbb{N}^*$, $Te, B, E1, E2 \in TL$ and $\gamma \in \sigma_g$,
$Eval_\gamma$: $TL_E \rightarrow T_F \cup \{\texttt{error}\}$ is defined by

(T1) $Eval_\gamma\ \texttt{no\_match} = \texttt{error}$

(T2) $Eval_\gamma\ x = \gamma\,(x)$

(T3) $$\frac{(Eval_\gamma\ Te)\ [\varepsilon] = c}{Eval_\gamma\ c^{-1}(Te) = c^{-1}(Eval_\gamma\ Te)}$$

(T4) $$\frac{Eval_\gamma\ Te_i = \texttt{error}}{Eval_\gamma\ f(Te_1,...,Te_n) = \texttt{error}}$$

$$\text{(T5)}\quad \frac{\forall\, i{=}1...n,\ Eval_\gamma\, Te_i \neq \texttt{error}}{Eval_\gamma\, f(Te_1,...,Te_n) = Rewr\,[f\,(Eval_\gamma\, Te_1,...,Eval_\gamma\, Te_n)]}$$

$$\text{(T6)}\quad \frac{Eval_\gamma\, B = \texttt{true}}{Eval_\gamma\, \texttt{if}\ B\ \texttt{then}\ E1\ \texttt{else}\ E2 = Eval_\gamma\, E1}$$

$$\text{(T7)}\quad \frac{Eval_\gamma\, B = \texttt{false}}{Eval_\gamma\, \texttt{if}\ B\ \texttt{then}\ E1\ \texttt{else}\ E2 = Eval_\gamma\, E2}$$

$$\text{(T8)}\quad \frac{\forall\, p{=}1...n,\ Eval_\gamma\, c(t_1,...,t_n) = Te}{Eval_\gamma\,(c^{-1}(Te))|_p = t_p}$$

Figure 42. Semantics of *Eval* for Domain TL_E

◊

Definition 33: *Application $Eval_\gamma$ for Domain TL_B*

For all $x \in X$, $f \in F$, $c \in C$, $p \in \mathbb{N}^*$, $Te, B, E1, E2 \in TL$ and $\gamma \in \sigma_g$,
$Eval_\gamma$: $TL_B \rightarrow \{\texttt{true},\texttt{false}\} \cup \{\texttt{error}\}$

$$\text{(T9)}\quad \frac{Eval_\gamma\, Te = \texttt{error}}{Eval_\gamma\, (\texttt{Generator}\ (Te) = c) = \texttt{error}}$$

$$\text{(T10)}\quad Eval_\gamma\,(\texttt{Generator}\ (Te) = c) = (\texttt{Generator}\ (Eval_\gamma\, Te) = c)$$

$$\text{(T11)}\quad \frac{(Eval_\gamma\, Te)\,[\varepsilon] = c}{Eval_\gamma\,(\texttt{Generator}\ (Te) = c) = \texttt{true}}$$

$$\text{(T12)}\quad \frac{(Eval_\gamma\, Te)\,[\varepsilon] \neq c}{Eval_\gamma\,(\texttt{Generator}\ (Te) = c) = \texttt{false}}$$

Figure 43. Semantics of *Eval* for Domain TL_B

◊

The evaluation strategy of c^{-1} and `Generator` is *call by value* (rules T3 and T9), whereas the `if-then-else` instruction has its usual non-strict semantics (rules T6 and T7). The generator inverse is not defined for zeroadic constructors, since we will never want to decompose an atomic value.

The compilation algorithm generates `if-then-else` tree structures, with `no_match` in the leaves corresponding to cases left undefined by the source axioms.

4.4.1 Object-oriented features of the target language

Our purpose is to be as general as possible in the choice of the target language. We want to use a minimal set of mechanisms in order to be largely language-independent and to make the prototyping process as intuitive as possible.

What we need is an object-oriented language with the following characteristics (See for instance [Wegner 87] for an explanation of object-oriented terminology):

- Abstract classes and deferred methods.
- Single inheritance, although multiple inheritance is no impediment.
- Inclusion polymorphism and dynamic binding.
- Polymorphic variables and heterogeneous data structures.
- Access to supermethod, for code reuse in general and for our validation by assertion mechanism.
- Encapsulation.
- Strong and static typing.
- Class attributes, by opposition to class members, are necessary to implement the protocol object mechanism. This is sometimes available through metaclasses in pure object-oriented languages.
- Restricting the visibility of some features to descendant classes only. This is useful for the realization of various kinds of auxiliary functions, be it at the specification level - *hidden operations*, declared in the `body` part of a CO-OPN module - or at the implementation level, for instance when dealing with memory-management.
- Non-primitive operations, i.e. non-redefinable functions, are necessary in cases where the generated code is deemed too sensitive to be redefined by the user. This might concern e.g. complex filtering algorithms for AADTs, or acute synchronization sequences in concurrent contexts. Note that programming languages do generally not prevent people from *redeclaring* functions, but these redeclarations will never be taken into account in OOMP since they are not visible from the abstract class.
- Overloading, in the sense of ad-hoc polymorphism, is needed since algebraic specification languages also offer this facility. Object-oriented languages usually include this possibility, although Eiffel constitutes here a notable exception.
- Genericity is useful for direct implementation of generic modules of the specification language.
- Exception handling capacity is profitable to cleanly handle situations where sufficient-completeness is not respected.

Except as noted, all of these features are available in main-stream object-oriented languages like Eiffel, C++ and Ada95.

4.5 The Compilation Algorithm

The compilation process consists mainly of a translation of pattern-matching into a procedural form. For this, we propose Schnoebelen's algorithm [Schnoebelen 88] adapted to OOMP. This choice is guided by the completeness of the algorithm: It is correct given the assumption that the rewrite system satisfies termination and confluence, and it is capable of compiling conditional axioms, as well as some other interesting extensions which we do not exploit here but might be useful in the future (subtyping and equations between constructors).

To tailor the compilation algorithm to OOMP, we just need modifying the code generation. We consider here only the case where the abstract code is evaluated in exactly the same environment as the concrete one, since they both implement the rewrite rules with identical semantics. We do not take into account the general case sketched in [Choppy&Kaplan 90], where it would be possible to have a general Horn-clause interpretor calling the compiled functions: The result of this is a powerful - but heavier - system capable of evaluating expressions involving unification instead of only simple filtering(2).

The function *Compile_For* compiles an *n*-ary operation *op* from its list *R* of defining rules, and from a non-empty set *T* of *n*-tuples of ground terms of T_C^n , which constitutes a description of its domain (or input arguments).

2. Informally, *unification* allows a two-way variable instantiation. For instance, if successful, the result of unifying two couples <x,t_1> and <t_2,w> is that x is bound to the term t_2 and w is bound to t_1. *Filtering*, on the other hand, does not allow the second couple to contain unbound variables, thus w would have to be a constant or an already instantiated variable in the frame of filtering.

Definition 34: *Compile For*

For all $x \in X$, $c \in C$, $k, p \in \mathbb{N}^*$, $B, E1, E2 \in TL$,
Compile R For: $T_C^n \rightarrow TL$ is defined by:

(C1) *Compile* Ø *For* T = `no_match`

(C2) if $T \subseteq G(\lambda_1)$:
Compile $(\pi_i => \lambda_i \rightarrow \rho_i)_{i=1...m}$ *For* T = `if` B `then` $E1$ `else` $E2$
where
$B = CompileRH_{\lambda 1}(\pi_1)$
$E1 = CompileRH_{\lambda 1}(\rho_1)$
$E2 =$ *Compile* $(\pi_i => \lambda_i \rightarrow \rho_i)_{i=2...m}$ *For* T

(C3) if $T \cap G(\lambda_1) = \emptyset$:
Compile $(\pi_i => \lambda_i \rightarrow \rho_i)_{i=1...m}$ *For* T =
Compile $(\pi_i => \lambda_i \rightarrow \rho_i)_{i=2...m}$ *For* T

(C4) if $T \not\subseteq G(\lambda_1)$ and $T \cap G(\lambda_1) \neq \emptyset$:
Compile $(\pi_i => \lambda_i \rightarrow \rho_i)_{i=1...m}$ *For* T = `if` B `then` $E1$ `else` $E2$
where
$B =$ `Generator` $((c^{-1}(x_k))|_p) = c$
$k.p =$ *choose* $(NMP_{\lambda 1}(T))$
$c = \lambda_1[k.p]$
$T' = \{ t \in T \mid t[k.p] = c \}$
$E1 =$ *Compile* $(\pi_i => \lambda_i \rightarrow \rho_i)_{i=1...m}$ *For* T'
$E2 =$ *Compile* $(\pi_i => \lambda_i \rightarrow \rho_i)_{i=2...m}$ *For* $(T - T')$

Figure 44. Rules for *Compile For*

The *choose* function of rule C4 is a function which chooses a minimal position inside any finite subset of $\mathbb{N}^*$. This function must be *regular*, which means that if $P' \subseteq P \wedge (choose(P) = p \in P') \Rightarrow choose(P') = p$. The *choose* function can e.g. simply take the leftmost-outermost position, although better heuristics exist [Schnoebelen 88].

The function $NMP_{t0}(T)$, the *non-matching positions* of the set T w.r.t t_0 , is defined as $NVP(t_0) - MP_{t_0}(T)$. $NVP(t_0)$ is the *non-variable positions* of t_0, i.e. the set of symbols not representing variables, whereas $MP_{t_0}(t)$, the *matching positions* of a ground term t w.r.t t_0, is defined inductively by

$\forall\, t_0 \in T_{F,X}, \forall\, t \in T_F$:
- $\varepsilon \in MP_{t_0}(t)$ iff $t[\varepsilon] = t_0[\varepsilon]$
- $p.i \in MP_{t_0}(t)$ (with $i \in \mathbb{N}$) iff $p \in MP_{t0}(t)$ and $t[p.i] = t_0[p.i]$

Finally, for $T \subseteq T_F$, $MP_{t_0}(T)$ is defined as $\cap_{t \in T} MP_{t_0}(t)$.

CompileRH is an auxiliary function which compiles the condition and the right-hand side of the current axiom into respectively the *B* and *E1* parts of rule C2.

Definition 35: *CompileRH*

For all $x \in X$, $c \in C$, $f \in F$, $k, p \in \mathbb{N}^*$, $B, E1, E2 \in TL$,
$CompileRH_\lambda : T_{F,X} \rightarrow TL$ is defined by:

(C5) if t is some $\lambda|_{k.p}$:
$CompileRH_\lambda\ (t) = (c^{-1}(x_k))|_p$

(C6) if $f\ (t_1,...,t_n) \neq \lambda|_p\ \forall\ p \in P(\lambda)$:
$CompileRH_\lambda\ f(t_1,...,t_n) = f(CompileRH_\lambda t_1,...,CompileRH_\lambda t_n)$

Figure 45. Rules for *CompileRH*

This concludes our presentation of the compilation algorithm. For its proof of correctness, termination and monotonicity, we refer directly to [Schnoebelen 88].

4.6 Low-Level Optimizations

As mentioned in the introductory description of OOMP (chapter 2), many usual optimization techniques, taken, in the case of AADTs, from the field of functional languages, may be applied in the generated abstract and symbolic classes:

- Common subexpression recognition consists in storing intermediate results in auxiliary variables in order to avoid multiple evaluations of the same expressions. This is a very important optimization, which is not performed by the given compilation algorithm, but the source paper states that it is not difficult to implement, since the concerned expressions are always internally structured as trees. It is also interesting to know that the compiler of the target programming language may do some of the work.
- "Remembering" the result of constant operations. By constant operations, we mean those without parameters. Since side-effects are not allowed in this context, functions must always return the same result. This idea is also applicable to the case of constant generators to implement *structural sharing*(3): In this case it is not the value, but the reference to the dynamic instance that will be retained.

3. *Structural sharing* denotes a memory management technique which avoids repetitive allocation and copying of dynamic data structures by reusing instead the pointers to the interesting structures. This technique is usually combined with reference counters in order to eventually recover the space occupied by entities which are no longer referenced by anyone.

- Elimination of tail-recursion in defined operations is possible and safe in the object-oriented paradigm because all the data structures are homogeneous (in fact they are declared heterogeneous, but at run-time they always will contain objects of the same class). This means that once a function has been selected through dynamic binding, it is not necessary to redispatch for performing recursive calls.
- Another application of the previous observation is that all recursive calls, not only the tail-recursive ones, can be transformed into static function calls (by opposition to dynamic binding), provided that the target language allows such distinctions. Mutual recursion (two functions calling each other) cannot profit from this optimization because it would lead to a strong coupling between these functions, hence disallowing redefining them separately.
- As mentioned in [Garavel 89], the compiler can recognize certain patterns in the definition of sorts, based on the profile of their generators. For instance, a sort having only constant generators can be translated into an enumerated type of the target language. Similarly, sorts with exactly one constant and one recursive generator can be directly mapped onto the predefined natural numbers of the target language. This optimization will lead to more compact representations of data types which otherwise would be implemented as linked data structures. It is important to notice that this technique does not allow recognizing specified functions in order to map them onto predefined operations of the target language: This is undecidable and is one of the motivations for the compiler to support mixed prototyping.

When applying the mentioned optimizations, there remains the condition that the internal representation be packaged inside an appropriate structure to stay compatible with the object paradigm.

4.7 Restrictions in Modular Specifications

The compilation algorithm at this point accepts all syntactically correct constructions of unstructured multi-sorted AADTs, except rewrite rules which perform pattern-matching on non-generator symbols, which are not left-linear or have equations between constructors.

Additional conditions must however be introduced, in relation with the notion of modularity. If we exclude the employment of generic modules and renamings, these restrictions may be classified in two groups as follows. For the syntactic and static semantic tests related to the use of genericity and renaming, we refer to [Flumet 95] and [Roques 94].

Well-Structured Algebraic Specifications

It may happen that an initially correct specification becomes inconsistent when enriched - i.e. imported - by a new module. To avoid this, we need to enforce some rules which guarantee the notion of *well-structured algebraic specification* [Bidoit 89].

Definition 36: *Well-Structured Algebraic Specification*

An algebraic specification *Spec* is *well-structured* iff it verifies the three following conditions:

1. Any module of *Spec* is the root module of a single sub-specification of *Spec*;
2. Any pair of distinct modules of *Spec* have disjoint signatures;
3. The generators are defined in the same module as the name of the sort they generate.

◊

The first condition is equivalent to the requirement that the dependency graphs of a well-formed CO-OPN specifications shall not contain any cycle (definition 15 of chapter 3). The second condition means that sort names and operations (identified by a name and a profile) shall be unique within the specification. The last condition states that one cannot add a generator to a sort outside of the module which defines the sort, and this includes the case where the new generator would be declared in the private part - the `Body` - of a CO-OPN module.

Correct Operation Definition

In the frame of rewrite systems, a defined operation $op \in OP$ is defined by a set of conditional rewrite rules $Rules(op) = (\pi_i => \lambda_i \rightarrow \rho_i)_{i=1...m}$, where λ_i are terms compatible with the profile of *op* (definition 28). In other words, each rewrite rule contributes to the definition of exactly one operation *op*. Moreover every operation must be declared in the interface before it is defined. Therefore, according to rule number 2 hereabove, the set of rewrite rules defining *op* cannot be scattered across several modules. In terms of CO-OPN syntax it gives the following definition:

Definition 37: *Correct Operation Definition*

An operation is *correctly defined* iff:

1. An operation is declared and defined in exactly one Adt module;
2. Each axiom of an Adt module corresponds to an operation declared in the interface of the same module;

3. To each operation declared in the interface of an Adt module corresponds at least one axiom of the same module.

◊

This definition is necessary in order to generate correct operation implementations. Rule 1 implies that an operation may be implemented exclusively on the basis of local information: It would be embarrassing to discover later additional axioms for an already implemented function. Rule 2 is obvious: There must not be any unused axioms in an implementation. The role of the last rule is to ensure that the generated functions are compilable: At the programming language level there must be a `return` statement somewhere which gives a result to the caller, and this result necessarily originates from the evaluation of an axiom.

4.8 Compilation of Algebraic Terms in Petri Nets

For translating the set of axioms defining an operation into a function of the target programming language, we follow the rules of Schnoebelen's algorithm [Schnoebelen 88]. The same algorithm can be exploited for compiling behavioural axioms during the implementation of algebraic Petri nets: According to the data flow established by analysis of the mode declarations(4) and the variable usage, each variable occurrence of the axiom will be considered as being either read from or written to. Therefore, the term each variable occurrence belongs to will be placed accordingly either in the left-hand or in the right-hand side of a fictive conditional rewrite rule. The compilation of this auxiliary rewrite rule produces the code which is necessary to access the variables correctly in presence of pattern-matching. Notice that in the case of CO-OPN, additional code must be generated for the control of the synchronization part of a behavioural axiom; more details are given in chapter 6. The idea of using rewrite rules to model the data flow in a Petri net was already exploited in [Choppy&Johnen 85]. They however applied it to simpler formalisms than algebraic Petri nets, and their objective was the simulation and not the compilation of Petri nets.

If we examine the *body* of the generated function `top` of figure 39, we notice that it is not specific to the defined operation it implements: It corresponds to a fictive axiom `match(push x on s)=x` the effect of which is to match a parameter *P* with the term `push x on s` and to return the value of the sub-term denoted by *x*. If for instance *P* has the value `push zero on empty`, then the above code will associate *x* with `zero`. The following piece of code implements exactly that behaviour:

4. To be compilable, the methods of a CO-OPN object are required to declare the mode `IN` or `OUT` of their parameters. The motivation for this choice is given in chapter 6.

```
1  procedure push_on_Match
2      (P: in Abstract_Stack;
3       Nat1: out Nat;
4       Stack1: out Stack;
5       Success: out boolean) is
6  begin
7    -- axiom: match (push x on s) = x
8    if Generator(Stack(P))=push_on then
9      declare
10       inverse : push_on_Arg := push_on_Inv(Stack(P));
11     begin
12       Nat1 := inverse.Nat1.all;
13       Stack1 := inverse.Stack1.all;
14       Success := true;
15     end;
16   else -- Error: P = empty
17     Success := false;
18   end if;
19 end push_on_Match;
```

Figure 46. Matching Function for the Constructor "`push_on`"

Compared to the previous code extract (figure 39), the profile now includes two copy-out parameters which return the components of the input parameter and a third which lets the caller decide what to do if the matching fails.

This mechanism may be exploited in many situations, for instance when an iterator must scan the contents of a place in an algebraic Petri net in order to find a token which has the required structure. For instance, if we have a place *P* which contains one or several stacks, then the precondition `P(push x on s)` may be specified for a transition, meaning that the precondition is satisfied if the place has a stack which is not `empty`.

The flexibility of this implementation scheme has a cost: Most of the time, the caller is not interested in all the parameters returned, hence an overhead for the unproductive computations. An alternative would be to generate a variant procedure for every specific pattern-matching needed, at the expense of losing generality. Another problem on a more pragmatic level, is to generate legible names for all these procedures.

In order to produce such pattern-matching procedures, the main modification from the original compilation algorithm is to generate assignments '`Success := false`' instead of raising exceptions in the cases where the pattern-matching fails. Formally, this would imply a change in the treatment of the value `no_match` of the target language *TL*, which we will not detail here.

4.9 Epilogue

In this chapter we have given the operational semantics of AADTs viewed as term rewriting systems, as well as the compilation algorithm needed for generating the initial abstract implementation required for the mixed prototyping process. Our contribution within this work is minor:

- First, we had to adapt an existing compilation algorithm to the frame of OOMP.
- Then, we showed briefly how to reuse this modified algorithm in the field of algebraic Petri nets.

The structure of the code which is actually generated on the basis of the compiler's target language *TL* will be detailed in the next chapter, since until now we have kept the discussion on a rather abstract level, without going into the practical problems of mapping whole specification modules into a particular programming language.

Chapter 5

Prototyping of AADTs

5.1 Introduction

The concept of *mixed prototyping* [Choppy 87] emerged in the context of algebraic abstract data types (AADTs) [Ehrig&Mahr 85]. The purpose of this chapter is to show how *mixed prototyping with object-orientation* (OOMP) applies to this formalism.

AADTs are used in a number of specification languages, such as PLUSS [Gaudel 85], ACT-ONE [Ehrig&Mahr 85] and the data part of LOTOS [ISO 88]. Many tools have been built to support this formalism: [Mañas&de Miguel 88], [Choppy 88], [Garavel&Turlier 93] and [Broy et al 93] are some examples. One of the reasons for this success is probably that the usage of algebraic specifications becomes relatively intuitive when interpreted as rewrite systems. They are then very close in spirit to certain functional or object-oriented programming languages. Another advantage is that when they are viewed as term rewriting systems, algebraic specifications may give way to very efficient automatic implementations in procedural or functional languages. In classical approaches, where the generated code is not intended to be read or modified, all kinds of optimizations may be applied without problem. However, in the frame of OOMP, legibility becomes important, not because the generated abstract classes are modifiable, but because their structure must be understandable and provide safe programming constructs at the developer's disposal for hand-writing new implementations.

5.2 General Mapping Rules

Our purpose is to provide for each construction of the source specifications an equivalent entity in the target language, the specific languages being respectively CO-OPN [Buchs&Guelfi 91] and Ada95. From now on, and unless noted otherwise, the name Ada will refer to the object-oriented version of the language [Ada 95].

The choice of the target language is determinant, in particular for what concerns the structuring primitives and philosophy. For instance, at the level of modules, in a purely object-oriented language like Smalltalk [Goldberg 84], there is no other structuring primitive than the class construct. On the other hand, in Ada, there are two separate constructs, the *package* and the *tagged record*. Another example is the relative independence, in Smalltalk, between compilation units and classes, whereas in Ada, there is no possibility of adding primitive operations to a class outside of its defining package(1).

In the case of Ada, the general results look like this:

- Each specification module is translated into a compilation unit, with one child unit per symbolic and user-defined concrete implementation. Child units have access to the private declarations of their parent unit. This is the way Ada arranges for the *protected* visibility of C++ [Stroustrup 91].
- Mapping (compared to the general class pattern of Figure 6 on page 19):
    1. Each specified sort is mapped into an abstract *tagged type* (the class construct of Ada) which is derived from the root class `ADT_Root`.
    2. The `constructor`s implement the generators of the sort, the `accessor`s are the set of generator inverses as well as the testor function called `Generator`, and the `operation`s are of course the defined operations of the specification.
    3. For each sort is generated a standard set of "predefined" functions, such as an equality operator(2)(3) and a primitive for writing a value to standard output.
- All the functions of the `constructor` and `accessor` groups are initially declared as abstract. The reason is that they are tightly connected to the internal representation of the class. They are the means by which the defined operations can be expressed, in the abstract class, in an implementation-independent way.
- A generic specification module must be directly represented as a generic Ada package.
- Renaming of sorts, generators and operations is more difficult to render, because the renaming clause is not as general as in CO-OPN: It is not just a textual trick in Ada, but has several semantic side-effects or restrictions which we will expose in the next section.

1. The constraints are even stronger than that: They are globally referred to as the *freezing rules*.

2. To use syntactic equality means that only *initial models* may be implemented. This corresponds to the standard usage in prototyping tools and is consistent with the fact that equations between constructors are not allowed (see e.g. [Garavel&Turlier 93]).

3. In the case of Ada, this operator must be redefined at each level of the inheritance hierarchy, because it does not follow the usual inheritance scheme: The Ada compiler will automatically insert low-level equality tests for any new fields of a class derivation, unless the programmer provides his own redefinition.

5.3 An Example in Ada95

In this section we present the details of the implementation scheme, as we designed it for Ada95. To begin with, we illustrate our words with a small example. Then, in the following sub-sections, we show how to treat the less canonical examples.

Note that in Ada95, functions returning an abstract type must themselves be abstract, which means that they can have no default implementation. To circumvent this, the return types are declared *class-wide*, i.e. of the polymorphic type enclosing all the descendants of the class:

```
type Abstract_Natural is abstract new Root_ADT with private;
function "+" (X,Y: in Abstract_Natural) return Abstract_Natural'Class;
```

To enhance legibility, the class-wide type is renamed into the name of the sort it implements (renaming of types in Ada may be performed by subtype declarations):

```
subtype Natural is Abstract_Natural'Class;
function "+" (X,Y: in Abstract_Natural) return Natural;
```

Another remark is that in the following examples, all algebraic abstract values are passed by value, even the result of constructors, which do dynamic memory allocation. This gives more homogeneous function profiles, and is closer to the spirit of AADTs and Ada than using pointers.

5.3.1 The Abstract Class

Suppose now we have specified and wanted to implement an Adt module of natural numbers. The compiler would then directly generate an abstract class `Abstract_Natural` and a derived class `Symbolic_Natural`.

The Abstract Class Interface

Given the following partial specification,

```
ADT Naturals;
INTERFACE
   USE Boolean;
   SORTS natural;
   GENERATORS
      zero : -> natural;
      succ : natural -> natural;
```

Figure 47. Partial Specification of Naturals

Then the following Ada package specification will be generated (first lines):

```
with Root_ADT_Pkg; use Root_ADT_Pkg;
with ADT_Booleans; use ADT_Booleans;

package ADT_Naturals is

     type Abstract_Natural is abstract new Root_ADT with private;
     subtype Natural is Abstract_Natural'Class;
     type Natural_Ref is access all Natural;

    -- Public class methods:
    ------------------------

    --/ Initialization routine for the selection of a concrete implementation:
        procedure Set_Natural_Prototype is abstract;

    --/ Show the prototype object:
        function Natural_Prototype return Natural;
```

Figure 48. Type Declarations and Class Methods for Naturals

The abstract class is where the information about the active derived class is stored and can be retrieved. OOMP does not only implement AADTs in terms of class hierarchies, but also provides mechanisms allowing more flexibility during the prototyping process, while keeping the advantages of strong and static typing. By using the abstract class as interface, and by resorting to polymorphism, we eliminate the need to modify or recompile client classes. We must however provide a means of designating the derived class to use as implementation, without mentioning its name in the client's code. That's the purpose of the prototype object.

The prototype object is typically used when it comes to creating *constants*, i.e. when constructing values from parameterless generators, such as `zero` for `Naturals` and `empty` for `Stacks`. The general rule is that the prototype object serves as a *controlling argument* for functions that otherwise could not be a primitive of the given class because it has no parameter of that type. For instance, the constructor function `zero`, specified as above, produces the following declaration:

```
function zero (Prototype: in Abstract_Natural)
       return Natural is abstract;
```

Figure 49. Declaration of Constructor Function `zero`

The argument `Prototype` is needed in order to tell the target compiler that this function is a primitive of `Abstract_Natural`. In other words, it can be redefined in derived classes, and the code that will be really executed is selected at run-time by inspection of the actual argument. A call to `zero` would look like this:

```
zero (Stack_Prototype); -- Ada95
```

or this:

```
Stack_Prototype->zero(); // C++
```

Figure 50. Invoking Constructor `zero` in Ada95 and C++

This may seem a bit awkward, but in reality this call can, in non-purely object-oriented languages, be wrapped inside a homonymous *class-wide* (i.e. non-primitive) function which hides the mechanism away from the clients (Figure 51). There is of course the cost of one supplemental function call to achieve this transparency, unless the target compiler is able to do inlining of subprograms.

```
function zero return Natural is
begin
   return zero (Stack_Prototype);
end zero;
```

Figure 51. Definition of Wrapper for Constructor Function `zero`

The rest of the package specification goes as this:

```
--/ Create a succ:
   function succ (Param: in Abstract_Natural) return Natural is abstract;

--/ Exceptions potentially raised by inverse of constructor 'succ':
   succ_inv_error: exception;

--/ Inverse of succ:
   function succ_Inv (Self: in Abstract_Natural) return Natural is abstract;
```

Figure 52. Declaration of Generator and Generator Inverse for Succ

As explained in the previous chapter, generator inverses are needed essentially for supporting pattern-matching. They are in fact accessor functions, and their use is tightly bound to another standard accessor, the overloaded `Generator` function:

```
--/ Generators of type Natural:
   type Natural_Generator is (zero, succ);

--/ Tell which generator created me:
   function Generator (Self: in Abstract_Natural) return Natural_Generator is abstract;
```

Figure 53. Declaration of Generator and its Associated Enumerated Type

Then come the predefined operations on the sort Natural: Some gymnastics is needed to use the “=” operator in Ada, but we think it is worth the trouble.

```
--/ Synt_Eq implements the test for syntactic equality
--/ Ada "=" operator must be redefined to Synt_Eq at each class derivation
   function Synt_Eq (Left,Right: in Abstract_Natural) return Standard.Boolean;
```

```
   function "=" (Left,Right: in Abstract_Natural) return Standard.Boolean;

--/ Print a natural on standard output:
   procedure Put (Self: in Abstract_Natural);
```

Figure 54. Some Predefined Operations

All the standard declarations are made; given the following set of specified operations:

```
OPERATIONS
   _ + _ : natural natural -> natural;
   _ - _ : natural natural -> natural;
   _ = _ : natural natural -> boolean;
   _ < _ : natural natural -> boolean;
```

Figure 55. The Defined Operations of the Specification

The profiles are generated like this (figure 56). Note that the equality operator above overloads the predefined syntactic equality, the codomain of which is “meta-boolean” and not the user-defined “boolean”, as explained in the previous chapter.

```
-- Specified operations:
------------------------

   function "+" (P1,P2: in Abstract_Natural) return Natural;
   function "-" (P1,P2: in Abstract_Natural) return Natural;
   function "=" (P1,P2: in Abstract_Natural) return ADT_Booleans.Boolean;
   function "<" (P1,P2: in Abstract_Natural) return ADT_Booleans.Boolean;
```

Figure 56. Declaration of the Specified Operations

Finally, the rest of the Ada package specification is the private part:

```
private

   --/ Initialize the class hierarchy:
      procedure Set_Hierarchy_Prototype (With_Prototype: in Natural_Ref);

   --/ Complete view of the type:
      type Abstract_Natural is abstract new root_ADT with null record;

end ADT_Naturals;
```

Figure 57. Private Declarations of the Abstract Class

The Abstract Class Body

For the body of the abstract class, we just give the implementation automatically generated for one of the specified operations, the addition. In the classical algebraic way of specifying

natural numbers, each value is expressed by a list having as length the number it represents. For instance, the term `succ(succ(succ(zero)))` is interpreted as the value `3`.

The specification of addition is defined by axioms deduced from the property of associativity, which, from an operational point of view, describes a recursive process where one of the arguments is progressively reduced to zero, while the resulting value is simultaneously incremented by one:

```
add-zero:           zero + y = y;
add-succ:           (succ x) + y = x + (succ y);
```

Figure 58. Specification of Addition on Naturals

In the abstract class, this is translated into the following (`Succ_Inverse(X)` being conceptually the same as taking the predecessor of `X`):

```
function "+" (X,Y: in Abstract_Natural) return Natural is
begin
   -- axiom add-zero:
   if Generator(X)=zero then
      return Y;
   else
   -- axiom add-succ:
      return Succ_Inverse(X) + Succ(Y);
   end if;
end "+";
```

Figure 59. Abstract Ada95 Implementation of Addition

This definition is as promised independent of the actual representation given to natural numbers: It works, provided that the derived classes implement the functions "`Generator`", "`Succ`" and "`Succ_Inverse`"(4).

5.3.2 The Symbolic Class

The abstract class provides an implementation of the axioms, but has no internal representation to put it to work. This is the role of the symbolic class. The internal representation is very standard: We use a variant record, with one alternative for each declared generator.

```
package ADT_Natural.Symbolic is

   type Symbolic_Natural is new Abstract_Natural with private;
   type Symbolic_Natural_Ref is access all Symbolic_Natural'Class;

--/ Call this in order to use this class as implementation:
   procedure Set_Natural_Prototype;

--/ Inherited abstract functions are now implemented:
   function zero (Prototype: in Symbolic_Natural) return Natural;
   function succ (Param: in Symbolic_Natural) return Natural;
```

4. A constructor function "`zero`" is also needed but it does not show in the given example.

```
        function succ_Inv (Self: in Symbolic_Natural) return Natural;
        function Generator (Self: in Symbolic_Natural) return Natural_Generator;
        function "="(Left,Right: in Symbolic_Natural) return Standard.Boolean;

    private

        type Symbolic_Natural_Variants (Tag: Natural_Generator := zero) is record
            case Tag is
                when zero => null;
                when succ => succ: Natural_Ref := null;
            end case;
        end record;

        type Symbolic_Natural is new Abstract_Natural with record
            variants : Symbolic_Natural_Variants;
        end record;

    end ADT_Natural.Symbolic;
```

Figure 60. Package Specification of the Symbolic Natural Class

The body of the symbolic class is trivial. Let us skip directly to a demonstration of what is possible to do with mixed prototyping.

An Optimized Symbolic Class

Figure 59 shows the direct translation of the recursive axiom `add-succ`; this is a rather expensive way of implementing things, especially when we observe that the result is nothing more than the concatenation of the lists representing the numbers `X` and `Y`. We could thus have a redefinition of addition exploiting this characteristic:

```
function "+" (X,Y: in Symbolic_Optimized_Natural) return Natural is
    Res: Symbolic_Optimized_Natural := Y;
    Cursor: Symbolic_Optimized_Natural := X;
begin
    while Generator(Cursor)/=zero loop
        Res := Symbolic_Optimized_Natural(succ(Res));
        Cursor := Symbolic_Optimized_Natural(Succ_Inverse(Cursor));
    end loop;
    return Res;
end "+";
```

Figure 61. Pseudo-code for an Optimized Symbolic Implementation of Addition

Figure 61 shows a non-recursive version of symbolic addition. The previous definition was tail-recursive, so the compiler could actually have optimized away that flaw. The internal representation of the new class is exactly the same as its ancestor's:

```
type Symbolic_Optimized_Natural is new Symbolic_Natural with null record;
```

Note that we might exploit in this redefinition the knowledge of the internal representation of `Symbolic_Natural` in order to avoid the calls to `Generator` and `Succ_Inverse`. That

would however “freeze” the internal structure of this branch of the class, disallowing any ulterior derivations. Figure 62 illustrates the new situation:

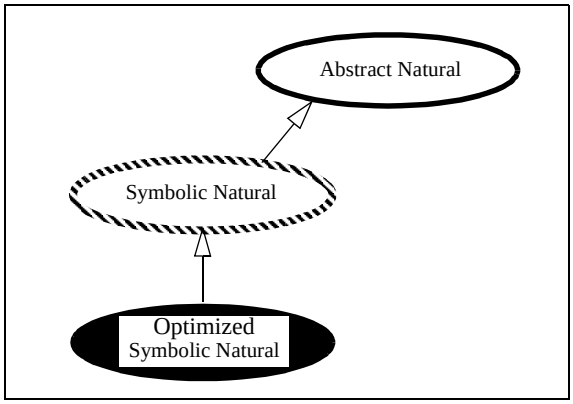


Figure 62. Specializing the Symbolic Class

Observe that we don’t modify the text of the original symbolic class, because it is generated automatically, and is as thus subject to accidental replacement by subsequent compilations. We create instead a sub-class having the same internal representation as the symbolic class.

5.3.3 The Concrete Class

Another, more interesting alternative to speed up the computations, is to choose an internal representation which takes advantage of the computer’s hard-wired treatment of natural numbers (using Ada’s predefined type `natural`). We would then create a concrete class where the definition of addition would look like this:

```
type Concrete_Natural is new Abstract_Natural with record
   Value: natural;
end record;

function "+" (X,Y: in Concrete_Natural) return Natural is
begin
   return Concrete_Natural'(Abstract_Natural with
                                Value => X.Value + Y.Value);
end "+";
```

Figure 63. A Concrete Implementation of Addition

Figure 64 below depicts the changes. Note that nothing prevents us from keeping the sibling classes, e.g. in order to step back to trusted implementations.

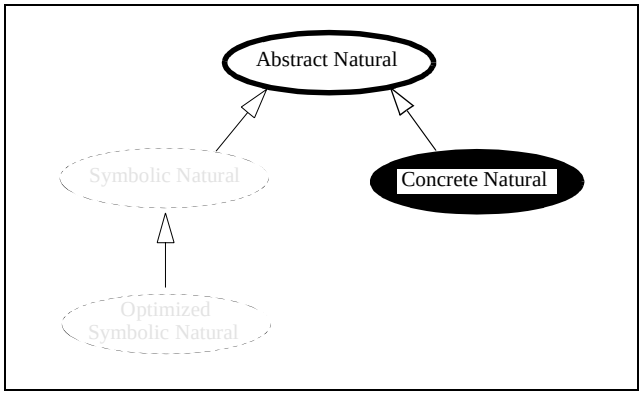


Figure 64. Incorporation of a Concrete Class

The interesting thing here is that the concrete class just has to provide a new internal representation, and define the associated low-level functions, i.e. the constructors and accessors. The functions corresponding to specified operations are stored in the abstract class and coded in an implementation-independent way such that they are not required to be replaced at once. This means that new internal representations can be tested before adapting the derived operations. Then, for instance, addition can be redefined using the corresponding machine-level instruction, while subtraction would be left until later, if not valued as critical for the prototype.

5.3.4 Implementation of a Concrete Class

As mentioned before, there is a minimal amount of work to do before a concrete class can be used. Let us review the implementation scheme in detail:

1. Provide an internal representation, i.e. instance variables such as the field `Value` in figure 63.
2. Implement the constructor functions, `zero` and `succ` in the case of `Natural`. The private fields (if any) of the ancestor `Abstract_Natural` are initialized with default values in the aggregate shown below.

```
function Zero (Prototype: in Concrete_Natural) return Natural is
begin
   return Concrete_Natural'(Abstract_Natural with Value => 0);
end Zero;

function Succ (Param: in Concrete_Natural) return Natural is
begin
   return Concrete_Natural'(Abstract_Natural with (Param.Value + 1));
end Succ;
```

Figure 65. Concrete Implementation of Constructor `succ`

3. For each non-constant generator, code the corresponding inverse. `Zero` is for instance such a constant generator: It has no sense to compute its inverse.

```
function Succ_Inv (Self: in Concrete_Natural) return Natural is
begin
   if Self.Value <= 0 then raise Succ_Inv_Error; end if;
   return Concrete_Natural'(Abstract_Natural with (Self.Value - 1));
end Succ_Inv;
```

Figure 66. Concrete Implementation of succ-inverse

4. Give the `Generator` function an implementation conforming to its intended sense, for instance:

```
function Generator (Self: in Concrete_Natural) return Natural_Generator is
begin
   if Self.Value = 0 then
      return zero;
   else
      return succ;
   end if;
end Generator;
```

Figure 67. Concrete Version of the Function `Generator`

5. Finally, each derived class must have a class method for the communication of the prototype object to the abstract class.

```
procedure Set_Concrete_Natural_Prototype is
begin
   ADT_Naturals.Set_Hierarchy_Prototype(Natural_Ref'(new Concrete_Natural));
end Set_Concrete_Natural_Prototype;
```

Figure 68. Providing a Prototype Object for Concrete_Natural

When the developer decides to use the concrete class as implementation, the configuration routine will simply call the associated `Set_Concrete_Natural_Prototype` procedure. The naming scheme `Set_`*`SubClassName`*`_Prototype` is important, otherwise the prototyping tool will not know how to initialize the class hierarchy(5).

5. Other approaches usually rely on the presence of a default constructor for the creation of prototype objects. We think our solution is cleaner: It supports the notion of opaque types, i.e. types which can be instantiated only with one of the explicitly provided constructors.

5.4 Special Cases of Mapping

In the previous sections, we examined how modules are implemented in the general case. Now we must find out how to treat problematic variants as well as advanced structuring primitives. Some situations may require awkward mappings: They correspond to modular decompositions of the system that are not compatible with object-oriented approaches.

In object-oriented languages, each primitive operation must be attached to a class. When this is not possible, we generally have several more or less artificial options. The most drastic one is to have the prototyping tool refuse the specifications that are not directly compliant with the philosophy of the target language.

It is however recognized that algebraic specification of AADTs lead to thinking in terms of specific data types and the operations they are to support, as we do in the object-oriented paradigm [Sommerville 92]. The specification process should therefore most of the time result in problem decompositions compatible with object-oriented implementations.

In chapter 2, we said that it could be beneficial to use an object-oriented design method in order to obtain interfaces which are compatible with the object paradigm. This remark applies especially to the two first cases described below.

5.4.1 Derived Operations Not Having the Sort of Interest in Their Profile

For the incremental redefinition of abstract functions by concrete ones to function properly, we rely upon the assumption that the sort of interest always will appear in the domain of the defined operations. This may not always be the case, as showed in the following example, where we want to define a hashing function for the specification of a hash table:

```
ADT HashTables;
INTERFACE
   USE Arrays,Naturals,Strings;
   SORTS hashtable;
   GENERATORS
      ...
   OPERATIONS
      hash : string -> natural;
```

Figure 69. Partial Signature of a Hash Table Specification

To transpose this operation into Ada95, we will have to choose between defining it as a class-wide function, in which case it cannot be overridden, and implementing it with an additional dummy parameter to make it a primitive of the class `Abstract_HashTable`. The latter solution was already described for the mapping of generators, in sub-section 5.3.1.

5.4.2 Modules Without Sort Definitions

Algebraic specifications allow enrichment of existing modules by simply adding new operations, but no sorts. For example, one could enrich the specification of `Naturals` with a definition of the factorial:

```
ADT Nat-Fact;
INTERFACE
   USE Naturals;
   OPERATIONS
      fact : natural -> natural;
```

Figure 70. Signature of Nat-Fact

There could be two ways to map this into Ada95:

- The first is to derive a class from the existing implementation of sort `natural`: This would be a class with only one additional primitive, namely `fact`, and having the same internal representation. This solution is not acceptable, since if several modules behave the same way, then there is no global criterion for deciding which concrete class should be used to implement the sort `natural`.
- The second solution is to define a package exporting a class-wide (global) function. This is very straightforward, but has the drawback that the function cannot be overridden.

5.4.3 Modules Defining Several Sorts

A good example of specification module with two interdependent sorts is the n-ary tree where both the leaves and the nodes have a name: It can serve for the specification of a Unix filesystem as in [Bidoit 89]. The module contains the sort `Tree` and the sort `Forest`.

```
ADT Tree-of-Info;
INTERFACE
   USE Info, Name-Set;
   SORTS tree, forest;
   GENERATORS
      < _ . _ >     : name forest -> tree;
      empty         : -> forest;
      _ plus _      : forest info -> forest;
      _ plus _      : forest tree -> forest;
   OPERATIONS
      name of : tree -> name;
      contents of : tree -> forest;
      name set of : forest -> name-set;
      name set of : tree -> name-set;
      son of _ named _ : tree name -> tree;
      son of _ named _ : tree name -> info;
      _ less the object named _ :
                        forest name -> forest;
```

Figure 71. Signature of Tree-of-Info

To represent this in Ada95, we will need one package with two mutually recursive tagged types. This is not a problem since the full type definition will take place only in the private part of the package: We do not need any kind of forward declaration.

After that, we will have to decide how to assign the generators and operations to the two classes. The rules are very simple:

- Constructors are naturally attributed to the sort of their codomain, i.e. to the class of values it generates. This means that "`< _ . _ >`" is a primitive of the tagged type `tree`, and the other generators belong to the type `forest`.
- The way defined operations are assigned to a class depends on the input arguments. This allows a straightforward mapping of observer operations, like `contents` shown above, which naturally belongs to the type `tree`. In the example of figure 71, this simple rule allows us to handle all the specified operations. If an operation has several possible candidates in its domain, then an arbitrary choice must be made.

5.4.4 Generic Modules

Does OOMP allow a homogeneous mechanism for the implementation of generic modules? In other words, can we implement generic modules of the source language as equivalent generic modules of the target language, and still keep the benefits of mixed prototyping? The answer greatly depends on how generic modules are instantiated in the source specification language. If instantiations can occur anywhere in the source text (call them *inline instantiations*), then it becomes harder to hide the inner workings for the support of mixed prototyping, such as the initialization of the prototype object.

In current versions of CO-OPN, inline instantiations are not allowed: A generic module may only be instantiated in a separate ad-hoc module which can, in turn, be shared by client modules. This ensures that the instantiation will be made at program startup, and the initialization of the prototype object can be performed as usual by the configuration routine.

5.4.5 Generic Parameter Modules

The notion of generic parameter module is often harder to reproduce in a programming language. Its purpose is to express that the actual parameter of a generic module should support a given set of primitive operations. One can even specify that these operations should respect certain axioms (see the theorem of figure 72), but this is beyond our objective: We have not examined if it is possible to automatically verify that the actual argument really honours the axioms of the parameter module. It may be feasible with a method similar to the dynamic testing described in section 5.7.

```
PARAMETER ADT OrderedElem;
INTERFACE
   USE Boolean;
```

```
    SORT elem;
    OPERATIONS
        _ < _ : elem elem -> boolean;
        _ = _ : elem elem -> boolean;
BODY
    THEOREMS
        (: Anti-symmetry :)
        (x < y) = true & (y < x) = true =>
            (x = y) = true;
```

Figure 72. Partial Definition of a Parameter Module

The challenge is to represent a parameter module by a construct of the target language. In Eiffel, we use the notion of *constrained genericity* in the header of the generic class: We express the fact that the actual parameter must be a descendant of the given class. This is the way we do it in Ada95 too.

In general, in languages with multiple inheritance, this is not an issue since any class can be derived from the combination of the intended actual parameter and from a class which supports the minimal set of operations needed for the generic entity. This kind of combination is called *mixin inheritance* and may be simulated in Ada95 by use of an auxiliary generic instatiation.

Remark that in C++ the semantic checks are made only at the time of generic instantiation: There are no constraints explicitly defined for the generic formal parameter, and a generic class cannot be directly compiled anyway. Therefore there will be no mapping of parameter modules in this language.

In the case of Ada95, it must be noted that there is a bias between the kind of parameter of the specfication, which is a module, and that of the target language, which is a tagged record. Ada95 also allows the definition of formal packages. This however requires additional formal functions and the language rules state that a primitive operation is no longer considered as primitive when seen through a formal function.

5.4.6 Renaming and Morphisms

Renaming and morphisms are used when instantiating generic modules. In the following example, the generic module `List` has a formal parameter module `OrderedElem` and is instantiated with the actual parameter `Naturals`:

```
ADT Nat-List AS List(Naturals);
MORPHISM
    natural -> elem;
    _ < _ IN OrderedElem -> _ < _ IN Naturals;
    _ = _ IN OrderedElem -> _ = _ IN Naturals;
RENAME
    list -> nat-list;
END Nat-List;
```

Figure 73. Specification of a Generic Module Instantiation

A morphism allows to express the exact correspondance between the entities used in the formal and the actual parameter modules. When using the mechanism of constrained genericity, the morphisms are entirely determined by the names and profiles of the primitive operations of the class: It is not possible to express the mapping explicitly. Again, this is due to the transposition of genericity into the object-oriented paradigm: The formal parameter is not a module but a class.

Renaming a type or a class in Ada is only possible through a subtype definition. Renaming a primitive operation gives a new equivalent function, but considered as non-primitive: To conserve the mechanism of dynamic binding, we must instead define a class-wide function which serves as a wrapper for the call to the original operation. Thus, given the following declaration:

```
_ < _ : natural natural -> boolean;
```

which gives this in Ada95:

```
function "<"(x,y: Abstract_Natural) return Boolean;
```

Then the following renaming in CO-OPN:

```
RENAME
   _ < _ -> _ inf _;
```

can be translated into Ada95 like this:

```
function inf (x,y: Abstract_Natural'Class) return Boolean is
begin
   return x<y; -- Dispatching call
end;
```

Figure 74. Mapping into Ada95 of the Renaming of a Defined Operation

This ends our presentation of the mapping schemes needed for translating the integrality of CO-OPN into Ada95. We have noticed that all kinds of structures may be implemented, but sometimes at the price of awkward constructs in the target language. This is mainly because the source language is not object-oriented whereas the target language is. Ada95 is a very rich language in the sense that it offers numerous constructs for structuring and composing modules, which means that probably no other programming language would be capable of such a performance. The advantages of Ada95 are its strong and static typing, its extensive support for overloading and genericity. Its block-structured syntax is also useful for managing the visibility of automatically generated identifiers. Finally, the fact that the concurrent and distributed aspects of the language have been standardized is an asset. Among the most obvious rivals, Smalltalk, Eiffel and C++, no one offers all of these characteristics. The more recent Java language [Niemeyer&Peck 96] could however consitute a serious candidate.

5.5 About the Reliance Upon Generator Inverses

In the field of algebraic specifications, the flexibility of mixed prototyping rests on the systematic definition and application of generator inverses. It may sometimes be complicated to find appropriate semantics for them, as shown in [Choppy et al 89], where a complete algorithm is presented to help with their design. This is especially true in presence of equations between constructors (See chapter 4).

It is not strictly necessary to sort out generators from other operations: They could be automatically identified, and the specification transformed in order to remove all equations between them [Comon 89]. Schnoebelen's compilation algorithm [Schnoebelen 88] is also capable of doing this. However, the need for generator inverses enforces the so-called *constructor discipline* [Guttag&Horning 78] during the specification, because generators must be considered as distinct from other operations in order to easily build the inverses. This gives place to a very imperative approach to specification, where constructors mainly serve as a means of selecting cases through pattern-matching, like in some functional programming languages, e.g. ML [Harper 86].

One could state that to distinguish generators from defined operations it is necessary to take premature decisions about the "representation" of the AADT. This is not really true, because the distinction is required only right before the compilation: Higher-level simulation tools do not need this information. This fact is emphasized in the algebraic specification language PLUSS [Bidoit 89], which offers three explicit completion states according to the level of achievement of the specifications: *sketches*, the most abstract category, *drafts*, where constructors are identified, and *specs*, for the final stage, where all the properties of the software have been expressed.

Mixed prototyping can live without generator inverses, but the alternatives are more constraining. Implementation granularity can be exploited: If the derived operations which need pattern-matching are implemented at the same time as the generators, then inverses are not needed [Choppy et al 89]. Another way is to impose a bottom-up approach to module implementation, and require the specification of an extensive set of observer and selector operations to compensate for the abolition of filtering [Roques 94].

Our purpose is however to take advantage of polymorphism in order to allow a finer implementation granularity, at the level of functions: Resorting to the systematic use of generator inverses is the most general technique.

5.6 Other Uses of Object-Orientation

OOMP resorts to inheritance and polymorphism as a means of switching between differing implementations of a same AADT. One can wonder if the object paradigm could also serve other useful purposes, and if maybe the mechanism described in this thesis excludes other possibly more meaningful applications of object-orientation. We will try to provide some answers in the next sub-sections.

5.6.1 Redefinition of Input-Output Operations

The prototyping tool provides a set of predefined operations in the implementation of AADTs, e.g. the equality operator and input-output primitives. The developer may tailor them if he deems this profitable.

Customizing Textual I/O

An abstract data value is printed out by default as the textual representation of the generators it is made of. The dual input operation can be implemented by a little parser capable of reading and analysing the output from the printing routine. Figure 75 shows an example where the class `Concrete_Natural` uses the default behaviour of I/O routines as defined in the abstract class.

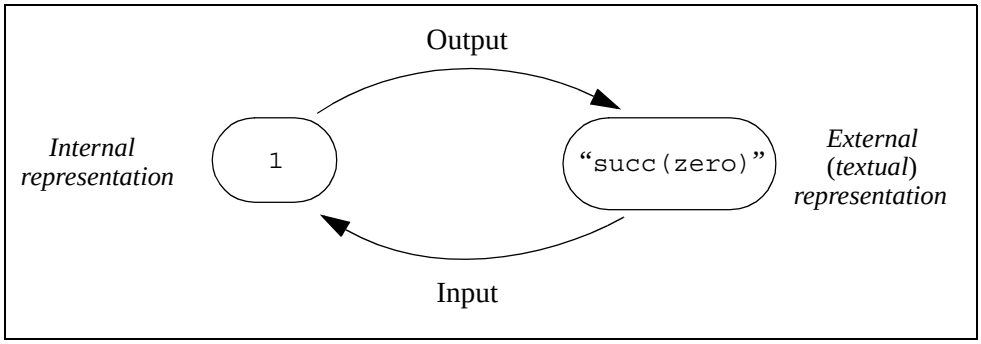


Figure 75. Default Textual I/O Routines used by `Concrete_Natural`

If these routines are redefined, it will usually be to produce more compact output, i.e. to write “1” instead of “succ(zero)”.

Customizing Marshalling Operations

Following the same principle as above, the user can customize the automatically generated *marshalling* operations (also sometimes called *flattening*), which allow communication of AADTs between two nodes in a distributed environment. One remarkable thing is that the

automatically generated routines can exchange *heterogeneous* data: By decomposing and transmitting values according to the structure of their generators[6], we define a communication protocol valid for any implementation of a given AADT[7]. By heterogeneous, we mean data from computers with differing architectures *and* AADTs implemented by distinct sub-classes. This capacity to overcome heterogeneity is quite natural in systems which treat data symbolically, and is explicitely mentioned as such in e.g. [Hulaas 95] or [Bergstra&Klint 96]. We could say that this technique constitutes a particular case of *heterogeneous hierarchical algebraic nets*, as formalized by Guelfi in [Guelfi 94]. The specificity of our point of view is that there is only one communication sort, the *stream of generators*, which is predefined and has the same implementation on all the nodes.

These routines may of course be redefined to better fit to the compactness of some concrete representation, but at the expense of no longer mastering heterogeneity. Also, to stay coherent, input and output routines must always be redefined pairwise, otherwise the communication protocol will be broken.

5.6.2 Subtyping

In algebraic specifications, subtypes are supported by the theory of *order-sorted algebras* [Goguen, Jouannaud&Meseguer 84]. Here, a subtype introduces *restrictions* w.r.t. its parent type, i.e. its carrier set is a *subset* of the one the parent is built on. This is useful for instance for cleanly expressing partial functions [Goguen&Meseguer 87]. This definition is however contrary to the usual notion of subtype in object-oriented languages, where a derived class also is considered a subtype of its parent class. Current main-stream languages offer only extension inheritance, which means that one can only add, and not retract, operations or components to the parent class. Certain programming languages, such as Ada, offer some support for subtyping in the algebraic sense, however by a mechanism that is parallel to the one of inheritance. Therefore the restrictive form for subtyping can only be simulated instead of directly represented - and enforced - in traditional object-oriented programming languages. A way to achieve this is to apply a *flattening* transformation as described for LOTOS in [ISO 88].

5.6.3 Coercion

Coercion is a means of reusing “code”, i.e. axioms, in algebraic specifications. An AADT can thus import the properties of another type and merge them with the ones it defines for itself, without there being any subtype relationship. Implicit type conversions are often supplied by the language. Assuming that the semantics of the imported operations are not

6. Remember that rewrite systems always lead to values in ground normal form. This is why we only need the generator symbols. The approach remains however valid in the general case where values can contain variables and operation symbols.

7. This can be compared to the idea behind the ASN-1 communication protocol.

violated by additional axioms[8], coercion can theoretically be implemented by code sharing. From an object-oriented point of view, this is feasible by the means of class derivation. But, considering that we should introduce no new hierarchical relationship in the implementation, coercion is only workable through the classical notion of type composition, or, in the worst case, by duplication of the original AADT.

5.7 Automatic Verification of Concrete Code

In chapter 2, we gave an overview of the techniques enabled by the OOMP class pattern which allow verifying that the concrete code respects the semantics of the specifications. In this section we will thus show how these techniques may be applied to the particular case of algebraic specifications.

Given the following pre-existing implementations of addition:

```
--/ Previous definitions:
function "+" (X,Y: in Abstract_Natural) return Natural;
function "+" (X,Y: in Concrete_Natural) return Natural;
```

The prototyping tool can then generate this function on demand:

```
--/ Testing version:
function "+" (X,Y: in Tested_Natural) return Natural is
   Result, Benchmark: Concrete_Natural;
begin
   --/ Call to supermethod:
   Result := Concrete_Natural(X) + Concrete_Natural(Y);
   --/ Call to abstract definition:
   Benchmark := Abstract_Natural(X) + Abstract_Natural(Y);

   --/ Check if results are equal:
   if Abstract_Natural(Result) /= Abstract_Natural(Benchmark) then
      raise Concrete_Implementation_Error;
   end if;
   return Result;
end "+";
```

Figure 76. Pseudo-code for Testing the Concrete "+" Operator

This scheme is independent of the internal representation, but necessitates a common object-oriented feature, access to the supermethods, i.e. calls to the overridden functions. This can be expressed either by renaming, as in Eiffel, or by a specific syntactic facility, as "::" in C++, or finally by some conversion mechanism such as the view conversions of Ada95.

8. Proving the conformance of new axioms is unfortunately undecidable.

The testing function is implemented as follows. We start by calling the concrete function, in case the execution should raise some exception: Thus the function will have the same observable behaviour as the one it replaces. Then we call the abstract version, using the concrete internal representation to get the correct value to compare with. After that, we have to perform the comparison using the abstract version of the "=" operator. We cannot use the user-defined equality operator, since it is not guaranteed correct.

The method has the advantage of not necessitating editing or recompiling existing code: When the developer wants to test his implementation, he notifies the prototyping tool, which in turn synthesizes an adequate sub-class.

This technique is nevertheless rather inefficient in its current realization since most operations of the abstract class are defined recursively, and the compiler may not always manage to derecursify the algorithms. Each recursive call will dispatch back to the testing function, before being properly redirected. This side-effect actually provides a more thorough testing mechanism, since all intermediate values are also controlled.

Other possibilities for testing the concrete code are:

- To use `theorems`[9] in CO-OPN specifications to express some properties of the operations that should only be exploited in the testing functions. This could lead to a more economic execution of run-time tests. If for instance an axiom expresses the commutativity of addition, it could lead to infinite recursion when implemented by a naive compiler. But declaring this property as a theorem could serve for testing. The following specification:

```
AXIOMS
   add-zero:     zero + y = y;
   add-succ:     (succ x) + y = x + (succ y);

THEOREMS
   add-comm:     x + y = y + x;
```

  Would produce something like this:

```
function "+" (X,Y: in Tested_Natural) return Natural is
   Result, Theorem_Result: Concrete_Natural;
begin
   --/ Perform intended operation:
   Result := Concrete_Natural(X) + Concrete_Natural(Y);
   --/ Check theorems:
   Theorem_Result := Concrete_Natural(Y) + Concrete_Natural(X);
   --/ Check if results are equal:
   if Abstract_Natural(Result) /= Abstract_Natural(Theorem_Result)then
      raise Theorem_Violation_Error;
   end if;
   return Result;
end "+";
```

- Have the user write his own pre- and postconditions, which would then only be exploited

9. In CO-OPN, `theorems` are formulas similar to `axioms` except that they express properties that are not intended to be interpreted.

in the testing class. To make this process systematic, the abstract class would have to define, for each operation, two initially empty boolean functions for expressing pre- and postconditions. The concrete class could afterwards redefine them as necessary. Note that the Eiffel language provides built-in constructs for invariant assertions and pre- and post-conditions.

5.8 Related Work in Compilation of AADTs

Several authors have already explored the process of compiling AADTs expressed by the means of algebraic specifications (see for instance [Kaplan 87] and [Schnoebelen 88]) and the algorithms are generally independent of the target implementation language. In actual tools however, the choice of a programming language is always dictated by some specific objective, and therefore they offer no continuity in the development process:

- When the purpose is to ensure a smooth transition from the specification formalism to the implementation language, *functional languages* offer the best alternative. For algebraic specifications to be executed efficiently, they must be *refined*, which means that operational information must be added. They tend by this process to become more functional than axiomatic in their nature. Therefore there is no paradigmatic leap between the source and the target language and it follows that the transition is easier to manage for humans. Another advantage is that the translation process is really straightforward, both environments offering the same level of abstraction. Functional languages also usually ensure safer semantics than other languages, and give better promises that the correctness of the formal specifications will not be violated by the implementation. This is the way followed by the KorSo project with the specification language Spectrum [Broy et al 93] that is to be translated into languages such as ML [Harper 86] and Haskell [Hudak, Peyton Jones&Wadler 92]. The disadvantage of this method is that the developer who wants to perform modifications has to cope with machine-generated code. Besides that, functional languages still aren't as well-accepted as supposedly more efficient procedural languages.

- Another obvious ambition can be to directly produce very efficient object code. The chosen target language is usually a *procedural* one like C, which also offers good portability. In this case the generated code is less legible, since cluttered by low-level details like compilation of pattern-matching and memory-management. The source and the target language do not provide comparable levels of abstractness, which implies that the code cannot readily be modified by humans who want to employ more clever algorithms or representations. The solution proposed in tools that compile the abstract data types of LOTOS, like Lola [Mañas&de Miguel 88] or Ceasar [Garavel&Turlier 93], is to incorporate compiler directives inside the specification modules in the form of comments, e.g.

"`extern`" or "`implemented by`". This presents the inconvenience that the two environments (of automatically-generated and hand-written code) cannot cooperate in a way independent of the internal representation of a data type. For instance, a function generated by the tool cannot work on an externally defined representation of the same data type. A final remark can be made about the type-safety of this kind of languages: they often do not support the notions of abstract data type and code reuse. In order to overcome this problem, current tools resort to generic pointers, and thus lose the benefits of static type-checking.

- Another trend, represented by the *Larch project* [Guttag, Horning&Wing 85], does not aim at synthesizing implementations of AADTs, but relies upon formal descriptions to check that hand-written code does not violate the specifications.

We are aware of no other approach of which the objective is to automatically generate AADT implementations in object-oriented programming languages.

5.9 Epilogue

In this chapter we demonstrated the applicability of OOMP to algebraic abstract data types. This was already done in [Choppy 87] in the original work on mixed prototyping. The frame of AADTs provides particularly interesting opportunities for exercizing mixed prototyping, because of the fundamentally recursive nature of its definitions. This characteristic has consequences on the implementation of the operations and on the data types too since the latter are defined by generators, which themselves are nothing else but operations which are not rewritten (or only into themselves in the case of equations between generators).

- The recursive type definitions always result in linked data structures when implemented automatically, and should often be replaced by more memory-efficient representations by the developer. This is especially true in the case of formalisms like algebraic specifications where state change is not allowed, and thus all defined operations induce a lot of copying. Another beneficial application field for mixed prototyping is when data structures are accessed randomly, i.e. not only sequentially. Giving them as representation a compact block of memory and redefining the indexing operator will then greatly speed up the computations. This concerns all kinds of vectors and tables, and also to some extent stacks and queues.
- The recursive operation definitions also systematically receive recursive implementations, which during the prototyping process may be converted into iterative routines in order to exploit the new representations of data structures and in general to be closer to procedural idioms. It is however true that the specification compiler should itself eliminate at least tail recursion.

The major constructs of the CO-OPN specification language have found equivalent constructs in a main-stream object-oriented programming language, namely Ada95. The principles remain however valid for any class-based language. We have tried to make the application of OOMP as intuitive as possible, by developing a direct mapping of the entities of the formal specifications to objects and operations of the target programming language. It must be noticed that some structuring primitives are difficult to translate elegantly and may even be impossible to implement or simulate in other languages than Ada95.

Finally, verification of the concrete implementations of AADTs is made astonishingly easy when mixed prototyping exploits the object paradigm. This may be explained by the purely applicative nature of algebraic specifications, which implies that there is no change of state, neither any possible side-effects. This remark will be confirmed in the next chapters, where the notion of state as well as concurrency will make the implementation and testing less immediate.

Chapter 6

Operational Semantics of CO-OPN Objects

6.1 Introduction

We showed in chapter 4 the operational semantics enforced for algebraic abstract data types (AADTs) in the frame of our *mixed prototyping with object-orientation* (OOMP) scheme. In this chapter our purpose is to do the same for the Object part of CO-OPN [Buchs&Guelfi 91]. In this language, the *Object* is the module construct which expresses the dynamic and concurrent properties of a system. To this end, CO-OPN extends algebraic Petri nets [Reisig 91] with a new notation and semantics for encapsulating and synchronizing sub-nets. A thorough presentation of the CO-OPN language was made in chapter 3: It is here that this information will be made profitable. Another semantics was briefly presented in [Buchs, Flumet&Racloz 93] in terms of centralized and sequential Prolog executions which allowed simulating models, whereas in this thesis the objective is to elaborate a modular interpretation of the semantics given in chapter 3. This is necessary for implementing CO-OPN specifications on distributed systems, and the obstacles are multiple. For instance, the semantics of CO-OPN relates an ideal world where all events are instantaneous and successul. The real execution environment is not so nice: Solutions must be searched for and unpredictable errors and delays may happen. Another constraint we assigned ourselves was to make the generated implementations prototypable according to the principles of OOMP. Therefore the code should reflect the structure of the specifications and provide a safe and comfortable framework for the developer who wants to ameliorate the implementation provided to him by the prototyping tool.

Our primary goal, when distributing the execution of a prototype, is not efficiency: We are not parallelizing sequential or centralized programs as a means of achieving speed-up. The purpose is rather to support a development methodology for systems which are by nature distributed, such as cooperative editors or automatic teller machines. This distinction probably constitutes the most salient difference between *simulation* and *implementation*.

We must also emphasize that these results are the product of an exploratory process which was to determine the feasibiliy of a distributed execution of CO-OPN prototypes. We do not guarantee that there is no redundancy in the techniques employed, and we do not claim that the algorithms are optimal or that they will provide very efficient executions. The approach has been essentially directed by the desire to answer for a maximal subset of the semantics of CO-OPN. We strived to reuse as much as possible techniques from diverse sub-domains of distributed systems: For instance by exploiting the concept of nested transaction, it became possible to benefit from existing results in fault-tolerance.

The operational semantics presented here are therefore not optimized nor formalized: We preferred to center the discourse on the requirements for implementing distributed prototypes and the design choices they induce in the architecture of the generated code. We think that formalizing and completely proving all the algorithms would require much more time and space than what could be allotted here. Finally, we are convinced that the complexity of the semantics of CO-OPN may be taken as an argument in favour of our didactic and somewhat verbose approach to the presentation of the algorithms.

The objective of this chapter is to incrementally show what characteristics of the CO-OPN language require new implementation schemes in order to enable distributed modular executions. To begin with, in section 6.2, we present the general implementation model. Then we will proceed by refining gradually the vision of the notion of event (section 6.3). First it will be seen from the outermost point of view, where it either fails or succeeds entirely. Then, by a first step towards the inner workings, we will establish how the run-time support of the distributed prototypes proceeds in order to simulate the hierarchical influence of the synchronization operators. Then we expose the distributed resolution layer (section 6.4), and finally the characteristics of the generated code (section 6.5). We conclude with some comparisons with other approaches (section 6.6) and a synthesis of the chapter (section 6.7).

6.2 General Implementation Model

The target systems for our prototyping methodology may be identified as the broad class of *client-server* systems (see e.g. [Mullender 93]). This category is made of distributed processes which communicate only by message-passing, i.e. they do not share a common memory. From a higher level point of view they can be considered as communicating by remote procedure calls (RPC) [Birell&Nelson 84]; in this context, the caller is denominated the *client* and the callee is the *server*. While serving a call, the server may have to request services from other processes and itself temporarily take the role of a client.

In distributed object-based systems, the processes may be simply called *objects*. This rejoins quite well the terminology of CO-OPN and our conception of a CO-OPN Object as being a single, possibly multi-threaded, process(1). The implementation of a *CO-OPN specification* is a system of distributed processes fully connected through a set of communication channels.

The execution environment is supposed *asynchronous*. This means that no hypothesis is made about the relative execution speeds of the different processes and that the communication delays are finite but unpredictable. The communication channels are *reliable*: Messages are not lost, duplicated or corrupted, and between any pair of processes the messages are delivered in the order they were sent. If this cannot be guaranteed by the hardware, then an underlying software layer may fulfill these requirements (see e.g. [ISO 84]).

A specificity of distributed systems is that processes may crash. To answer for this property, we exploit the concept of *nested transaction* [Moss 81] which ensures that the effects of a process failure do not spread across the whole system. We think that resilience to failures is a must for prototypes which aspire to be used as end-user products, by opposition to simulators or throw-away prototypes for which the only requirement is fidelity to the specifications. By the way, it should be emphasized that an implementation is not supposed to correct the errors due to its specification. In particular, a deadlock originating from the specification will be inevitably reproduced by the prototype.

Another fundamental characteristic of distributed systems is that no observer has access to the global state of the system. This is due to the fact that there is no shared memory and that there is no single clock which might give to the processes a common time reference for detecting causality relations and for synchronizing. To cope with this deficiency, messages will be relied upon for all data transfer as well as for supporting the algorithms which maintain as much as possible a coherent vision of time [Lamport 78]. An important constraint is then to avoid overloading the communication channels by finding a good balance between the accuracy and the quantity of information conveyed by the messages.

6.2.1 A Simple Example

Let us show a very simple example of distributed execution in order to demonstrate the kinds of messages exchanged. In the following figure, Object o_2 is stabilizing. An internal transition t of Object o_2 wants to synchronize with method m_1 of Object o_1. This method in turn wants to synchronize with method m_0 of Object o_0. The call to m_0 succeeds, the call to m_1 also terminates successfully, and finally the transition t succeeds, therefore the whole

1. In practice, this may lead to granularity problems, since CO-OPN Objects are usually rather small. Current work in the CO-OPN/SANDS group aims at providing original means of configuring and coordinating groups of Objects [Buffo&Buchs 96].

synchronization succeeds. All Objects can then stabilize, starting with `O0`, and finally `O1`. When `O1` is stable, `O2` can continue the stabilization it was already pursuing.

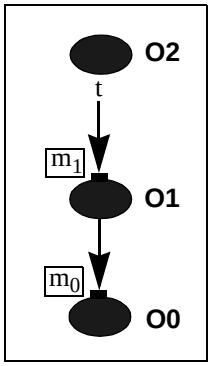


Figure 77. A simple Example

In terms of messages, this execution produces the following:

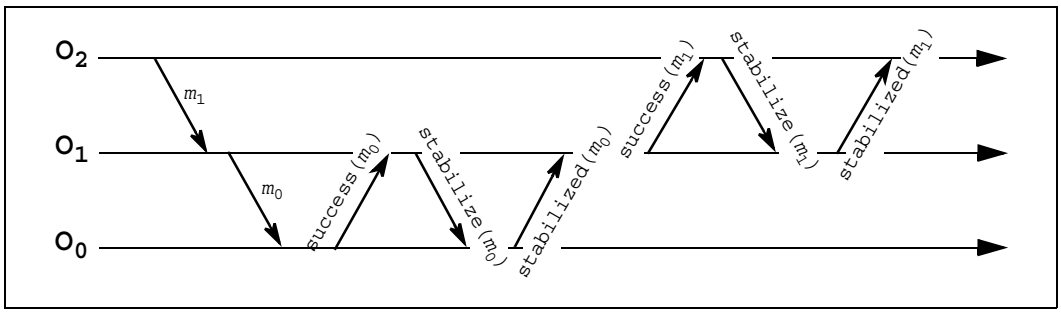


Figure 78. Messages Exchanged for the Execution of the Simple Example

We can notice in particular that every message refers to a specific method call m_0 or m_1 (this is a temporary way of denoting the dependency between messages which will be adjusted later in the chapter). Another information given by this figure is that the emitter of the synchronization, `O2`, also controls the subsequent stabilization; this is performed from the lowest Object and upwards.

6.2.2 Structure of the Generated Prototypes

In the generated prototypes, each CO-OPN Object consists of five layers (Figure 79):

- A message-passing service. This would preferrably be an asynchronous RPC mechanism [Walker, Floyd&Neves 90], even though the communication model of CO-OPN is syn-

chronous. Asynchrony is necessary for performing method calls in parallel, as required for the implementation of the simultaneity operator of CO-OPN. The advantage of an RPC layer compared to a simple message-passing service is that the remote procedure calls are seen as normal, local procedure calls. It transparently marshals the arguments and performs the adequate dispatching inside the target process in order to select the right procedure to execute. If additionnally this may be performed directly by the target compiler, as for Ada95 [Ada 95], then it relieves the prototyping tool from a significant burden.

- The low-level concurrency control layer: It must guarantee that each method call behaves as an atomic (*transaction-oriented*) action and takes into account the nature of the synchronizations requested (i.e. simultaneity, sequence or alternative).
- The resolution layer, which manages a Prolog-like search for a local state which allows serving incoming synchronization requests. This local state must also be compatible with the global state of the system, which implies that distributed backtracking may occur.
- The model structural description, which is the result of an automatic transcription of the structure of the specifications into the target programming language. This layer records for instance the dependencies between places and transitions. It also serves as an interface which renames and overloads many lower-level primitives for the application layer.
- The application layer, where the modelled CO-OPN Object is described by translation of the source behavioural axioms into an object-oriented representation. Since the evaluation of the guards is performed here, it means that this layer also contributes to concurrency control.

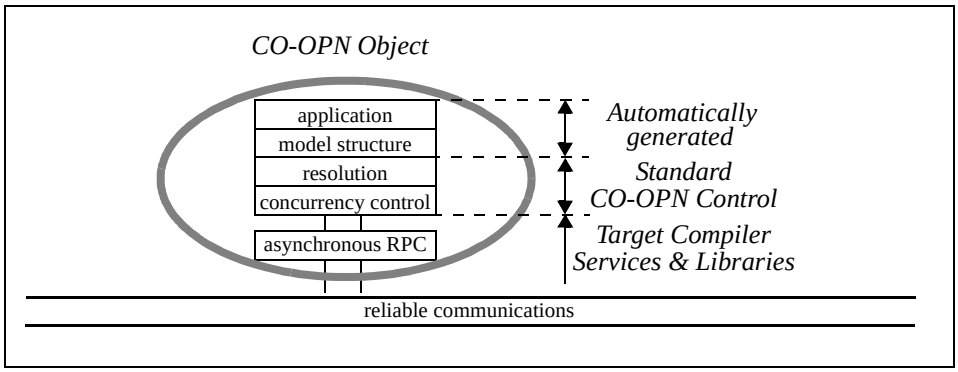


Figure 79. General Prototype Architecture

Our prototyping scheme is to be exploited only on the application layer, which we also call the *action class* in [Buchs&Hulaas 95], since the developer should not have to bother with the lower-level layers, which are very specific to the semantics of CO-OPN. The code generator could, in future work, propose a set of standard implementations for the lower levels, among which the user would choose the most adequate according to his global knowledge of the model: Intrinsic parallelism, presence of shared objects, or need for non-

determinism are some of the factors that might be combined. The lower levels are collectively referred to as the *control* class (see figure 80):

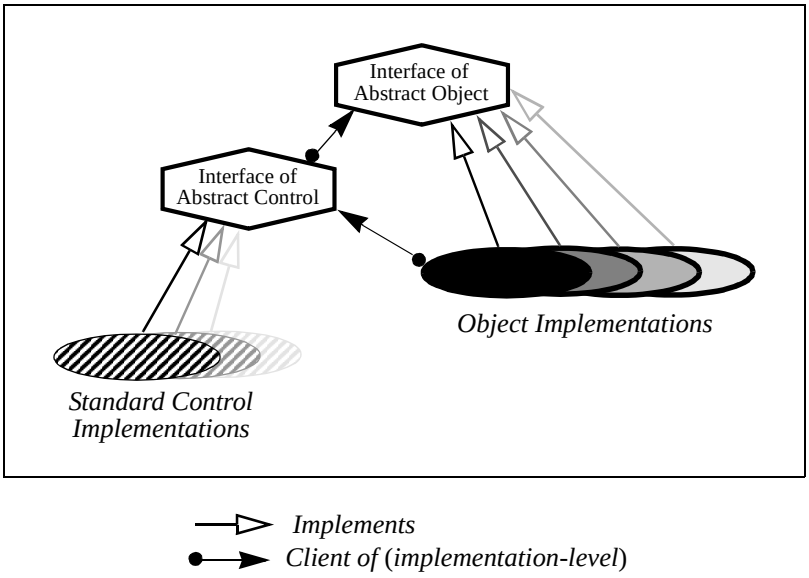


Figure 80. CO-OPN Object Implementation Model

There is thus a clear distinction between *control* and *action*. As depicted on figure 80, the control part sees only the interface of the action class, which must therefore be considered as a *black box*. In other words, the control is *orthogonal* to the functionality, something which is vital for mastering the cumulated complexities of concurrency control and resolution mechanism. This is also what determines the freedom and the limits of the prototyping process.

It is of course compulsory for the prototyping tool to warn the user when a specification is not compilable. All criteria must be based on a static analysis techniques: It would be incoherent for a running implementation to suddenly halt because it encounters an unsupported situation. At the same time, it is desirable to have a prototyping tool which compiles a maximal subset of the source language. Our approach is to accept nearly the complete language as input and to design the implementation so that the most frequent situations are handled the most efficiently. For instance, the execution should involve virtually no overhead in the circumstances where non-determinism is not needed.

In conclusion, we designed the control part of the CO-OPN prototypes to be modular and completely distributed. The motivation is on the one hand, that CO-OPN Objects are naturally distributed since their local states are encapsulated[(2)] and they communicate only by

method calls, and on the other hand, because we want to promote pairwise reuse of specification and implementation modules.

6.2.3 Environment of a Distributed Prototype

The notions of *environment* or *frontier of a specification* do not exist in the CO-OPN formalism. This is how we designed the connection between both worlds in the implementation:

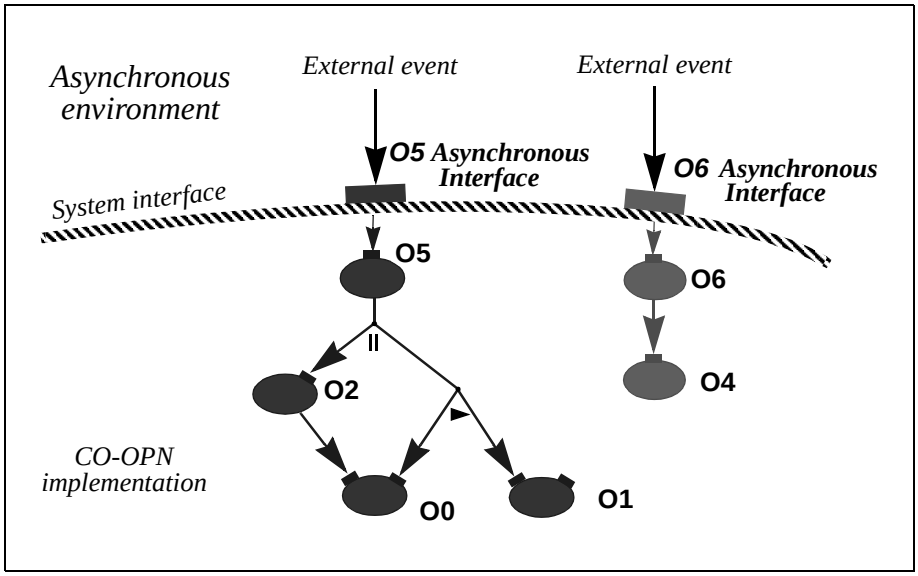


Figure 81. A CO-OPN Specification and its Environment

In the above figure, the frontier of the system is called the *system interface*. All the roots of the directed acyclic graph formed by the static Object dependencies have an *asynchronous interface* which makes the system available to the asynchronous outside world and which converts all events into CO-OPN transactions (to be defined later).

The Synchronous/Asynchronous Interface

The CO-OPN world views time as a succession of discrete instants. One of the roles of the asynchronous interface could therefore be to discretize the continuous time scale, but this is not really useful since the execution of our prototypes is not driven by any notion of clock or time limit, compared to reactive systems [André et al 96] [Boniol&Adelanto 93]: The synchrony of CO-OPN is not an implementation constraint, but rather just a way of elimi-

2. Apart from the notion of *stability*, as seen in chapter 3.

nating interleaving-based non-determinism. The interfaces can therefore simply serialize the external events which are transmitted to them.

An interesting extension could however be to allow clients of the asynchronous world to benefit from the advantages of strong synchrony for the execution of certain events. This would require the distributed system interfaces to coordinate in order to make these events appear as simultaneous to the CO-OPN implementation, as in [Caspi&Girault 95].

In our prototypes, the main role of the interface is to signal to the asynchronous clients, by the means of a call-back mechanism, when a method becomes firable. This is needed in order to override the purely hierarchical organization of the specifications: We do not want the client to continuously poll the exported methods. In comparison, within the CO-OPN implementation, the strictness of the hierarchy is relaxed by the principle of stabilization which propagates events downards as well as upwards.

A final purpose for the interface could be to embed CO-OPN synchronizations within an external transaction. Even at the lowest level of interaction, this raises some practical problems such as the compatibility between the different formats of transaction identifiers. We have not had the time to deepen this aspect of the prototypes.

Resemblance of CO-OPN Implementations and Simulators

One characteristic, which should be emphasized already now for the sake of clarity, is that within the distributed prototype, all actions must be considered as temporary. This is due to the non-determinism, which implies that no decision may be considered as definitive until the top-level event, i.e. the one which was transmitted to the prototype through the interface, is successful. This has several consequences on the prototype:

- There is no connection with the real “physical” time: Any decision can be retracted, which means that time can sometimes be seen as going backwards. This is similar to the notion of *virtual time* in optimistic discrete event simulators (see [Ferscha 96] for an introduction). The term *optimistic* denotes the fact that the treatment of events is started before it is certain that they should really happen, hence the presence of a *rollback* mechanism for correcting the situation when necessary.
- There is no input or output to a terminal or other physical device before the top-level event is successful because such operations are usually irreversible. We need ad-hoc mechanisms which tell Objects when it is safe to perform actions which cannot be undone.
- All intermediate states during the treatment of an event are accumulated in memory until the complete success or failure of the top-level event. This can sometimes give raise to storage problems.

The main computation within the prototype is performed in a closed world, without interaction with its physical environment. This actually makes the implementation very close to a simulator. The difference lies essentially in their respective purposes. A distributed simulator is usually designed for raw speed: There is no concern for maintenance or extensibility of the software. Moreover, a simulator will often eagerly search for all possible behaviours, whereas an implementation, at least in CO-OPN, has a passive attitude and is idle between two inputs.

6.3 The Concurrency Control Layer

6.3.1 Representing Synchronizations as State Diagrams

In order to show the control structures needed for managing synchronizations, we will base ourselves on the modular proof diagrams developed in section 3.6.8. These control structures correspond to the activity of the emitter of a synchronization: The receiver Objects have a more passive but at the same time more flexible behaviour since they never know whether they will take part in the rest of a given synchronization. That is why these diagrams would be too rigid for modelling the control of the receiver Objects. Instead they have the same set of states (modeled by ovals), but the given transitions are not taken into account.

6.3.1.1 The Basic Synchronization

Let us start by the *basic* synchronization, i.e. synchronizations without any sequence, simultaneity or alternative:

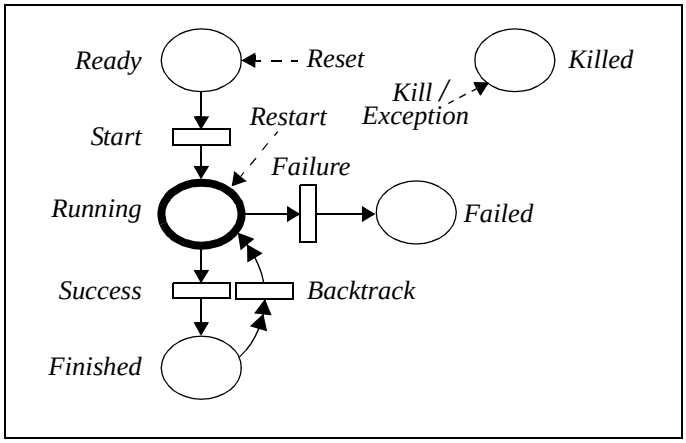


Figure 82. State Diagram for the Basic Synchronization

After the state *Finished* there are some additional states not shown here which are related to the establishment of a distributed coherent state (see subsection 6.3.2.5)[3]. A common property among all state diagrams to be presented in this section is the fact that at any moment a `Kill` or an unhandled `Exception` in the action part may lead to the unrecoverable state `Killed`, and that a `Reset` may lead back to state `Ready` from anywhere but from state `Killed`. A `Restart` is equivalent to a `Reset` directly followed by `Start`. Finally, notice that the thick circle of figure 82 is the interchangeable part among the different synchronizations as we will see below. These diagrams are used to indicate the state of progress of each synchronization and are referred to from the internal structure shown in appendix A.4 on page 234.

A complete list of supported messages is given in appendix B on page 237.

To *Backtrack* means to jump back and reevaluate the method or stabilization corresponding to the circle pointed at by the double headed arrow. This implies that the associated method call or stabilization is non-deterministic, i.e. that it can furnish several answers or Object states (i.e. *markings*) for the same source state. When an operation is non-deterministic in this sense, we say that it constitutes a *choice point*: It encloses an iterator which enables it to compute a new answer each time execution returns to it by backtracking. If, by backtracking, control jumps backwards beyond a choice point, then this choice point is *reset* to its initial state, so that the next time it will again deliver its first answer as if it were the first time execution passed through it. If a choice point has given all its solutions, it fails and execution returns to the immediately preceding choice point, still by the means of the backtracking mechanism. We will generally use the term *to retry* to denote the action of reevaluating a non-deterministic operation for obtaining new answers.

Backtracking is performed each time a method call returns the status `Failure`. If there are no more methods or stabilizations to reevaluate, then the current part of the synchronization ends in the state *Failed*, and control returns to the previous choice point, which is the enclosing synchronization. If the latter has its origin in another Object, then we will use the term *distributed backtracking* in contrast with normal backtracking which happens when there still exist choice points to exploit in the current Object. If there are no more enclosing synchronizations at all, it means that the top-level event cannot be fired in the given state of the system, and thus the whole computation can be abandoned.

6.3.1.2 The Sequential Synchronization

The diagram presented in figure 83 corresponds to a serial evaluation of sequential synchronizations. It would be conceivable to do this in parallel, but we think that it would lead to too many conflicts, and thus backtracks. In the future it could be interesting to identify suffi-

3. Adding these special states would make our basic state diagram identical to the one given for nested transactions in [Moss 81], except for the possibility of reset, restart and backtrack which is specific to our design.

ciently loose Object interconnection topologies where parallel evaluation of sequences would lead to good performance.

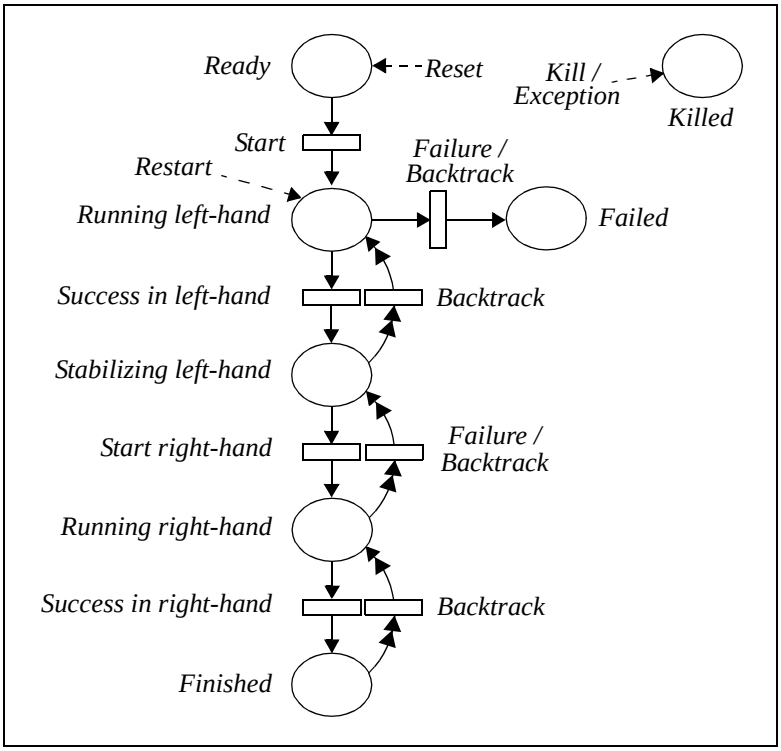


Figure 83. State Diagram for the Sequential Synchronization

We mentioned in chapter 3 that in CO-OPN a stabilization is always successful. It could then seem strange, as in figure 83, that it is possible to backtrack from a stabilization. This does in fact not mean that a stabilization may fail *per se*, but rather that, during a reevaluation, it may be unable to produce *another* stable state. In other words, all its choice points have been exhausted.

6.3.1.3 The Simultaneous Synchronization

The diagram presented here allows parallel evaluation of simultaneous synchronizations.

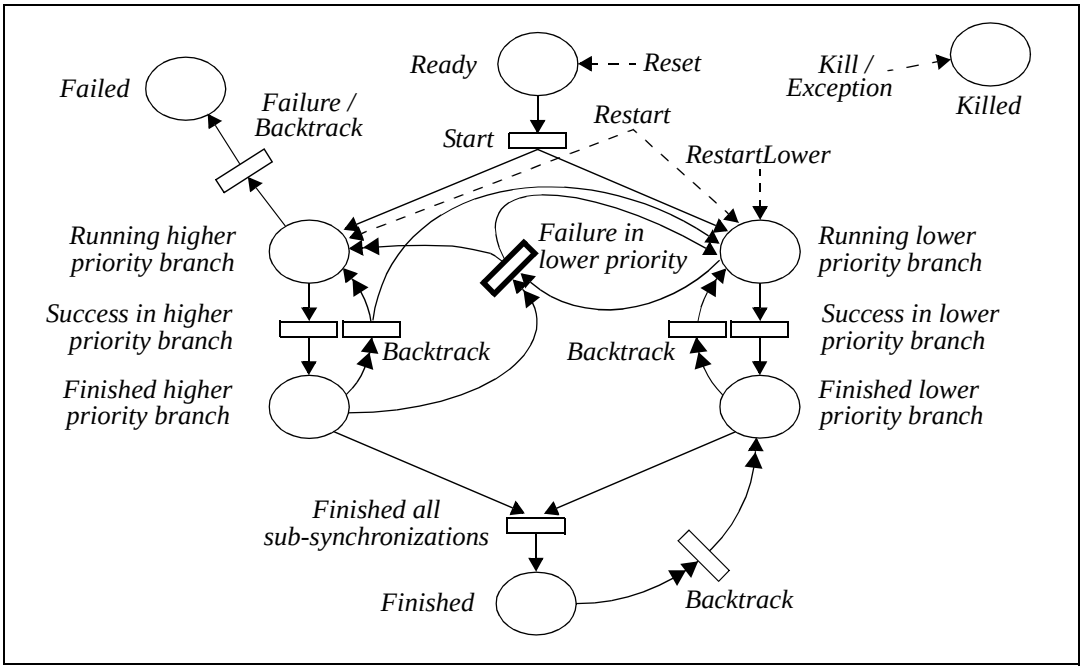


Figure 84. State Diagram for the Simultaneous Synchronization

The relative priorities of the different branches of a simultaneity are established in accordance with the total order in the Object dependency graph: This will be explained in section 6.4. The thick rectangle of figure 84 has two input arrows: This indicates that both of the two corresponding conditions are to be fulfilled for the transition to take place, as in Petri nets. More concretely, if the lower priority sub-synchronization returns `Failure`, it is restarted only when and if the higher priority branch has finished successfully. Another information given by this figure is that the lower priority branch is restarted (because of the single headed arrow) each time the higher priority part is retried by local decision, and the same effect may be obtained for solving conflicts in remote Objects by sending the message `RestartLower`.

6.3.1.4 The Alternative Synchronization

The diagram presented here allows parallel evaluation of alternative synchronizations. although it will have to be performed sequentially when an Object is shared by several branches of the alternative.

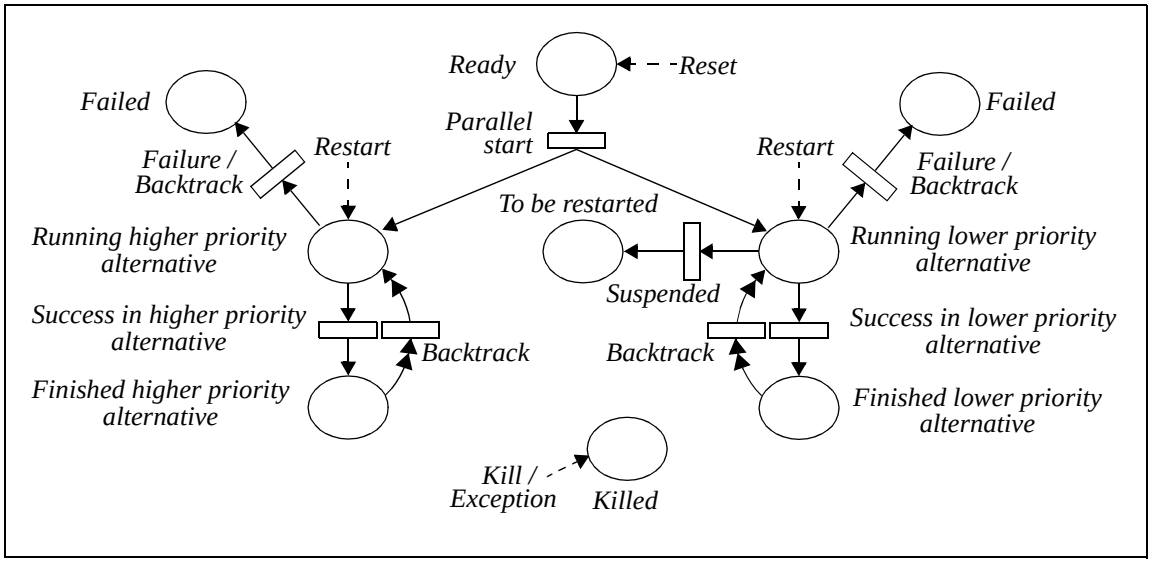


Figure 85. State Diagram for the Alternative Synchronization

The relative priorities of the different branches of an alternative are established according to their textual order in the specification, but the developer may change this by prototyping at the programming language level. The purpose of the priorities is for Objects shared by several alternatives to decide which one to favor. The lower priority one is `Suspended`, and the emitter of the alternative receives a corresponding message so that it may `Restart` this branch if the higher priority alternative fails. Since the suspension is performed at the level of the shared Object, a `Restart` message may be sent directly to that Object instead of restarting the whole branch from scratch.

Finally only one branch of the succeeded alternatives will be kept, and is then roughly to be considered as a basic synchronization, as in figure 82.

6.3.2 Events viewed as Nested Transactions

The notion of nested transaction, which we will introduce in this section, has several utilizations in the CO-OPN implementation. It serves as a concurrency control mechanism, as a vector for the transmission of global knowledge about the synchronizations, and as a guarantee for fault-tolerance.

6.3.2.1 Failure Atomicity of the CO-OPN Model

As explained in chapter 3, all events in the CO-OPN formalism are *atomic*, i.e. they have an *all-or-nothing* behaviour. This was already true for classical Petri nets, and has been extended to include the notion of synchronization within CO-OPN. Another reason for

seeing events as atomic is that they are described by *predicates*, i.e. they state some relation between their parameters and local variables. If it cannot be arranged for this relation to hold, then the system is left untouched, as if nothing had happened. Method calls in CO-OPN may be nested, and each nesting level has this same property. If the evaluation of a method call fails at some sub-level it does not mean that the whole event must be aborted: In case of don't know non-determinism the execution will try another alternative. It is only if the top-level predicate fails that the corresponding event is considered as impossible. This vision of atomicity induces that events may quite naturally be mapped into the notion of *nested transactions* [Moss 81] on the operational level.

By extension, we can even state that atomicity is inherent to the logic paradigm. Therefore, predicates may transparently support fault-tolerant executions: The developer does not have to provide explicit information for the underlying system to identify the range of an atomic action, and to handle appropriately cases of failure. This idea was first introduced in [Guerraoui et al 92] in the field of object-based distributed systems. In previous approaches, explicit information was required at the programming language level (e.g. the *Argus* language [Liskov&Scheifler 83] or the *Arjuna* class library [Dixon et al 89]). In the logic paradigm all-or-nothing atomicity is part of the computation model.

6.3.2.2 Concurrency Atomicity of CO-OPN Implementations

The notion of strong concurrency in CO-OPN implies that all events which are not sequential must be considered as happening exactly at the same moment on a discrete time scale (see chapter 3). There are two sources of concurrency, the first one being the concurrency specified by the use of the simultaneity operator, and the second one being the concurrency due to independent events. As we will see all along this chapter, strong concurrency is unnatural in asynchronous systems and must therefore be enforced artificially by the run-time control of each Object. This induces a cost that we would rather avoid when possible.

We must obviously respect the form of strong concurrency originating from the use of a simultaneity operator. We may however try to circumvent the concurrency of independent events by acting as if they were fired at different moments: Since they have no causal relationship, we may as well place them on two adjacent "ticks" of the time scale. This flexibility is not expressed directly by the semantics of CO-OPN, but there is room for it in the fact that the language does not specify what the environment and frontier of a specification consist of and where the events originate from.

Let us define the notion of independence; for this definition we will before need to establish that a *participant* in an event is an Object, of which a method or transition has been fired during the event. The *top-level* or *root* participant in an event e is the Object which has the highest number (according to the total order $\sqsubset$) among all the participants in e.

Definition 38: *Independence of Events*

Given *Spec* a CO-OPN specification constituted of a set of Objects *O*, $A \in Mod(Spec^A)$ an algebraic model, and two events $e_1, e_2 \in \mathbf{E}_{A,M(Spec),O}$, then the events e_1 and e_2 are *independent* if their respective top-level participant Object are distinct. ◊

Now that we have decided to fire all independent events at their own instant, it would be preferrable not to be too restrictive about this serial order in order to benefit from the parallelism which can be exploited as long as the respective sets of participants are disjoint. In order to guarantee that the concurrent treatment of two independent events does not lead to inconsistencies in the Objects' states, and in particular that intermediate states are not disclosed, they must be serialized at the level of the common participant Objects. This is called *isolation* or *concurrency atomicity* and is another feature of atomic transactions.

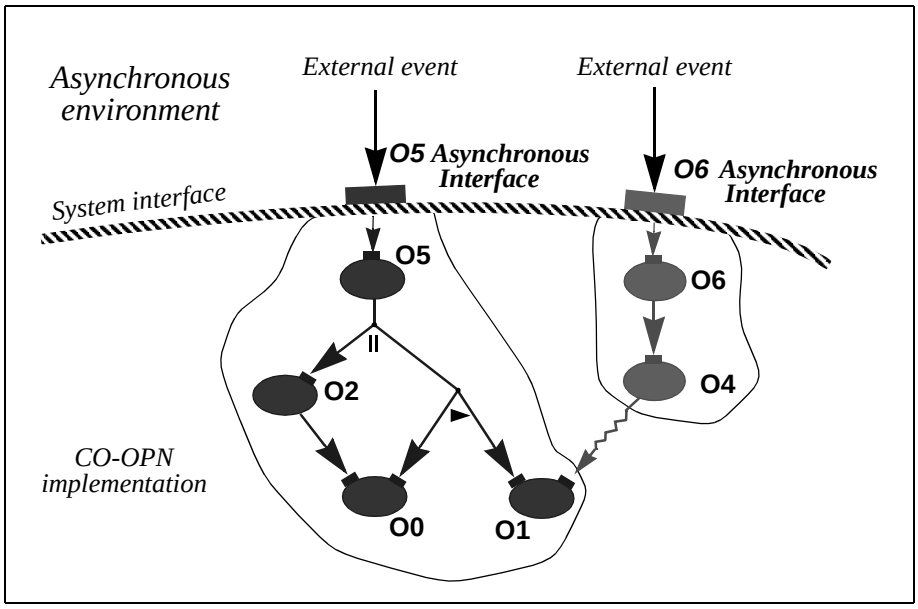


Figure 86. A CO-OPN Implementation with Isolation of Transaction Trees

In the situation depicted here, Objects `O5` and `O6` are waiting for the results of their current event, the treatment of which is protected by an "envelope" which prevents for instance `O1` from being spuriously accessed by `O4`.

6.3.2.3 Origin and Characteristics of Nested Transactions

Transactions have become a basic principle for preserving coherence in distributed databases in presence of concurrency and failures [Bernstein, Hadzilacos&Goodman 87]. Initially, transactions had a monolithic structure, forming single blocks of sequential opera-

tions. Nested transactions are an extension of classical "flat" transactions which provide better support for parallelism and fault-tolerance. This communication model has progressively been integrated into concurrent object-oriented programming languages (for an overview we refer to [Guerraoui 95]) and into general-purpose libraries for distributed environments, such as CORBA [OMG 95]. Roughly, fault-tolerance is guaranteed by the fact that each transaction brings the system from one coherent state to another. These special states are saved to stable storage, and consitute a safe base to return to whenever a node crashes(4). Even in normal conditions it may happen that a transaction fails for instance because the request (or event) it conveys does not obtain the needed resources. To take this into account, each object has a rollback mechanism which allows it to undo the effects of the uncompleted transaction(5). A transaction which terminates successfully is said to be *committed*, otherwise it is *aborted*, whatever the reason.

The relationships between nested transactions are tree-oriented. Therefore we will use a related terminology. Transactions having no subtransactions are called *leaf* transactions. Transactions having subtransactions are called *parents* and their subtransactions are their *children*. Similarly, we will use the terms *ancestors* and *descendants*, also as reflexive relations. *Superiors* and *inferiors* are respectively the non-reflexive versions of ancestors and descendants. The *top-level* transaction is a transaction which enters the system at a top-level Object. We will also use the terms *ancestors*, *descendants*, *inferiors* and *superiors* for (dynamic) Object relations. For instance, in the situation depicted by figure 86, `O1` is an inferior of `O5` but not (yet) of `O4` or `O6`.

Transactions have been defined as sequences of object requests which satisfy a set of "ACID" properties [Harder&Reuter 83]: (*A*) *all-or-nothing*, (*C*) *consistency*, (*I*) *isolation* and (*D*) *durability*. It is the duty of the underlying run-time support to ensure the *A*, *I* and *D* properties, while the application programmer ensures the *C* property (as far as he can, i.e. wouthout having to cope with failures or concurrency). This is the coherency contract between the programmer and the transaction management software in classical database systems. It was showed to be inappropriate for nested transaction systems in [Guerraoui 93], and another version was proposed, the contract defined in terms of "N-ACID" properties. We propose to slightly adapt this latter contract to the case of CO-OPN: The "C-ACID" properties are different in the sense that the concurrency which appears between dependent subtransactions is part of the specification, and not an optimization of a sequential execution. The *CO-OPN-atomicity* of CO-OPN transactions are described by the following:

4. We do not discuss here the low-level mechanisms by which failures are detected and recovered from. Relevant information can be found e.g. in [Chandra&Toueg 96] and [Verhofstad 78].

5. For an operation to be "undoable" it must not interact with the physical world. For instance, it is usually not possible to undo output to a terminal. Operations which are irreversible must be delayed until it is certain that the current event succeeds globally. Interestingly, CO-OPN does not provide any I/O primitives.

- (*C-A*) *CO-OPN all-or-nothing*: Either a transaction is committed and has its desired effects, or it is aborted and has no effect.
- (*C-I*) *CO-OPN-isolation*: The intermediate states of the objects manipulated by a transaction are neither visible to concurrent siblings nor to independent transactions.
- (*C-D*) *CO-OPN-durability*: The effects of a committed top-level transaction and those of its committed descendants are not undone by a failure.

If *CO-OPN-atomicity* is guaranteed for all transactions, then each execution will appear to happen as if each transaction tree executes alone in the system, starting from a state produced by a sequence of committed transaction trees. Then the replies and final state of each transaction tree will depend on the computation inside the tree, which is the responsability of the developer since it is precisely the behaviour specified by the CO-OPN sources.

- (*C-C*) *CO-OPN-consistency*: When it executes from coherent initial states, a transaction tree produces coherent replies and coherent final states.

When the developer exercises the freedom given to him by incremental prototyping, he must remain faithful to *CO-OPN-consistency,* the other properties being guaranteed by the control layers of the prototypes (Figure 79).

We have not yet seen by which mechanism isolation between transaction trees is ensured: That is the role of the locking protocol.

6.3.2.4 The Locking Protocol

Classical transaction systems have been the subject of enormous research in order to increase the performance by interleaving compatible transactions in shared objects. This has resulted in several definitions of the notion of *serializability* (see e.g. [Weihl 89]), the role of which is to provide a criterion for deciding when two transactions can be interleaved instead of executed in sequence. For instance, two requests may be candidates if the elementary operations they are made of, such as read and write, are commutative.

In CO-OPN, the elementary operations which make up methods (and thus the transactions) are token insertion and withdrawal from places. By static analysis of the behavioural axioms, it would be possible to determine, for each Object, sets of methods which may be interleaved instead of executed in strict sequence. We decided not to explore these possibilities because we have the feeling that the stabilization process which is activated in the middle of each sequential synchronization (see chapter 3) would systematically invalidate these interleaved executions.

The standard *two-phase locking protocol* (2PL) [Eswaran et al 76] is designed in the following way: There are two distinct phases, the first being when execution proceeds

downwards and extends the transaction tree by including (i.e. by locking) more and more objects, and the second phase is the upwards movement, where locks are progressively released (but the corresponding objects are not really freed until the global commitment which we will describe below). In order to guarantee serializability it states that once an object has released its locks, it cannot aquire new ones. For implementing CO-OPN this semantics would be too restrictive and anyway we do not exploit the notion of serializability: This means that the latter requirement does not need to hold, resulting in a *single-phase locking protocol*.

Another simplification, compared to classical database systems, may be introduced because CO-OPN is based on Petri nets, where events can only *modify* Objects, i.e. they do not provide any means of *reading* token values without before withdrawing them from their place. All Object accesses are therefore done in mutual exclusion, and we will not have to provide shared locks for read-only operations.

Definition 39: *CO-OPN Transaction*

A *transaction* in a distributed CO-OPN implementation is a mechanism which encapsulates each event in order to guarantee that concurrency and failures will not interfere with a correct execution. To each component of an event corresponds a subtransaction of the top-level transaction. Each transaction has a system-wide unique identity, the *Xid*. ◊

It should be emphasized that the term *components of an event* is here taken in the syntactical sense of page 54, which means that every branch of a synchronization tree is a subtransaction with its own identity. On the following figure, transaction `T55` is an example of subtransaction which represents an intermediate branch of the synchronization tree. `TI55` and `TI66` are top-level transactions(6).

6. We use this symbolic naming convention: Each subtransaction name is made of the number of the creator Object followed by the number of the target Object. Sometimes an index is used if several subtransactions exist between a given pair of Objects. An initial ‘I’ means that the subtransaction actually started from the interface of the given creator Object.

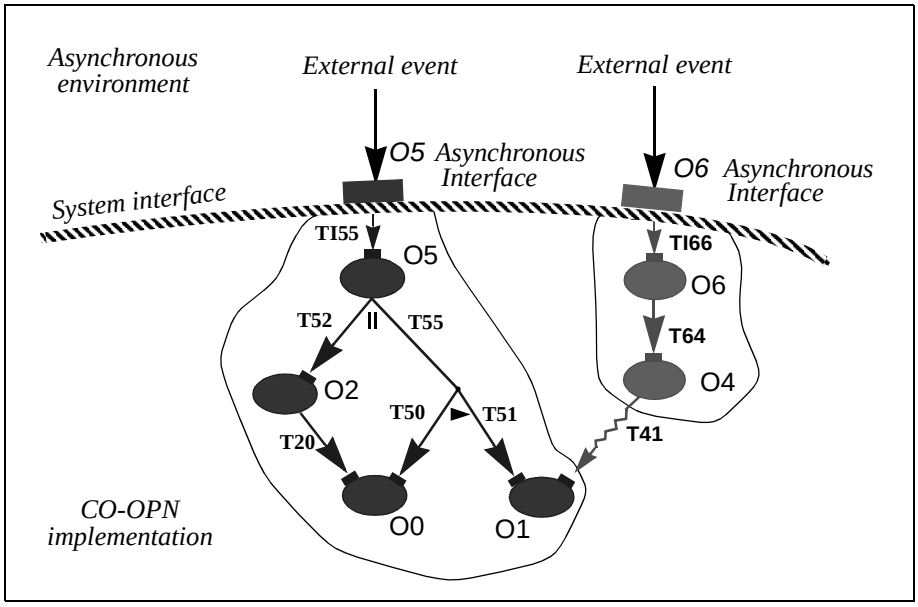


Figure 87. Transactions and Subtransactions in a CO-OPN Implementation

An Object creates a subtransaction for every event it is involved in. To each subtransaction is associated a copy of the Object's state, which is the state to restore if the subtransaction aborts. If the subtransaction encapsulates a method call, then we call *target* Object of the subtransaction the Object who will execute the method call.

In our approach, each transaction identifier is synonymous with a lock identifier: The target Object identifies the lock by the *Xid* of its owner. This means that each transaction locks at most one Object. We say that a transaction *holds* a lock when the target Object is effectively locked by this transaction.

Definition 40: *CO-OPN Locking Rules*

i) A transaction may hold a lock on an Object if the Object is free or if the other lock holders on the same Object have a common ancestor with the requesting transaction; otherwise the requesting transaction is blocked until one of the two mentioned conditions is fulfilled.

ii) When a transaction commits, its parent (if any) inherits a copy of its lock as well as the ones it inherited from its own committed inferiors.

iii) When a transaction aborts, its lock is discarded, as well as the locks it inherited from inferiors. Its superiors (if any) still keep their locks.

◊

The point in making the parent inherit the lock, i.e. the *Xid*, of a committed transaction is to transmit to the top-level transaction the list of all committed subtransactions. This list is needed for the atomic commitment protocol which will be briefly described later.

The *Xid*s are system-wide identifiers created independently by each Object. One way to realize this is to use the identity of the creator and to add some locally unique number to it. The identity of the creator Object may for instance be the number assigned to it in relation with the total order $\sqsubset$ [7]. It is also convenient to include the identity of the target Object, in order to know directly from the structure of a *Xid* the list of participants in a transaction.

The following describes how the *Xid* of the top-level transaction is built: The subtransactions create their own *Xid*s by *piggybacking*, i.e. by concatenating, a local subtransaction *Id* to the *Xid* of their parent [Moss 81]. The term *lock history* is sometimes used to refer to the accumulated *Xid*s of all superiors.

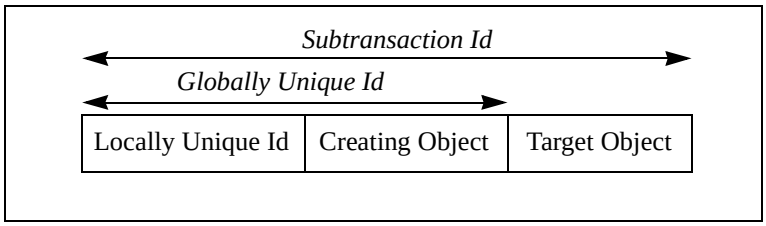


Figure 88. Structure of a Subtransaction Id

In the previous example (figure 87), subtransactions `T21` and `T53` have the follwing *Xid*s:

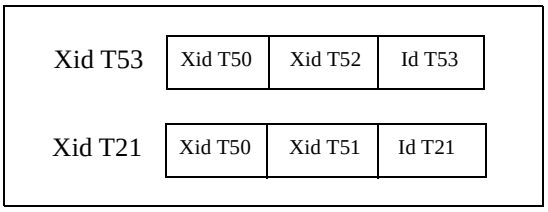


Figure 89. Structure of Two Subtransaction *Xid*s

It is then easy to see that the whole transaction tree may be reconstituted locally by any Object which has a copy of all the *Xid*s of the leaf subtransactions. In particular, it is possible to determine if some Objects are shared between several branches of the tree. This property is used by each Object in order to find out if an incoming transaction is indepen-

7. If no such number or identity is available, then the address of the Object must be directly used: In a Unix/ Internet network, this would be the IP address of the node, the Pid (process id) and possibly a Tid (thread or task id).

dent from any current lock holder, in which case it will be blocked until the current transaction tree terminates.

6.3.2.5 The Two Phase Commit Protocol

Once the useful computation of a transaction has been performed, the Objects involved must decide if they can consider their current state as coherent and save it to non-volatile memory. After this is done, the Objects may release all the locks which have accumulated on them during the transaction, and start to serve pending requests from independent transaction trees. This is the role of the *atomic commitment protocol*.

More precisely, the utility of the atomic commitment protocol is to ensure the above-mentioned *all-or-nothing* property, also called *failure atomicity.* This is realized by leading all participants to agree on whether the transaction is to be committed or aborted. By running this protocol atomically, it is possible to notice node crashes at any point of its execution and avoid that incoherent states are installed by the participants.

These are the properties that must be fulfilled by the atomic commitment protocol [Babaoglu&Toueg 93]:

- AC1: All participants that decide reach the same decision
- AC2: If any participant decides `commit`, then all participants must have voted `yes`
- AC3: If all participants vote `yes` and no failures occur, then all participants decide `commit`
- AC4: Each participant decides at most once (i.e. a decision is irreversible)

The de facto standard is the *two phase commit protocol* (2PC) [Gray 78]. In this protocol there is a single privileged Object, the coordinator (usually the top-level Object, or its interface to the asynchronous world in the case of CO-OPN), which controls the execution:

1. The coordinator sends the message `RequestVote` to all participants. Each participant decides locally if he wishes to commit or abort the transaction:
   - If a participant decides to commit, he replies `yes` and saves the new state to permanent memory; both the old and the new states are now safe. Then he waits for the decision of the coordinator.
   - Otherwise he replies `no`, and aborts the transaction by undoing its effects.
2. The coordinator collects the replies (`yes` or `no`) from the participants. If all participants have answered `yes`, then he decides to commit. If a single `no` is recorded or if a failure is detected then he decides to abort.
3. The coordinator notifies its decision to all participants which have not replied `no`. According to the decision, each participant either deletes the old state in permanent memory and definitively adopts the new state, or it deletes the new state in permanent memory and keeps the old state as current state.

The weakness of this protocol is that it is both centralized and blocking, which may be desastrous if the coordinator crashes during the execution of the protocol. Some other solutions have been proposed [Skeen 81],[Guerraoui&Schiper 95], but it is not our purpose to develop further this discussion, as the choice of an atomic commitment protocol is independent from the locking protocol used and does not involve any particular design choices in the generated prototypes.

In appendix E, Figure 123 on page 270, an example of 2PC execution is demonstrated in the frame of the collaborative diary example.

6.3.2.6 Deadlock Avoidance

We have seen that the isolation property is guaranteed by locking Objects and blocking independent transactions. This may however lead to deadlocks if there is no global criterion for deciding which transactions are to have highest priority.

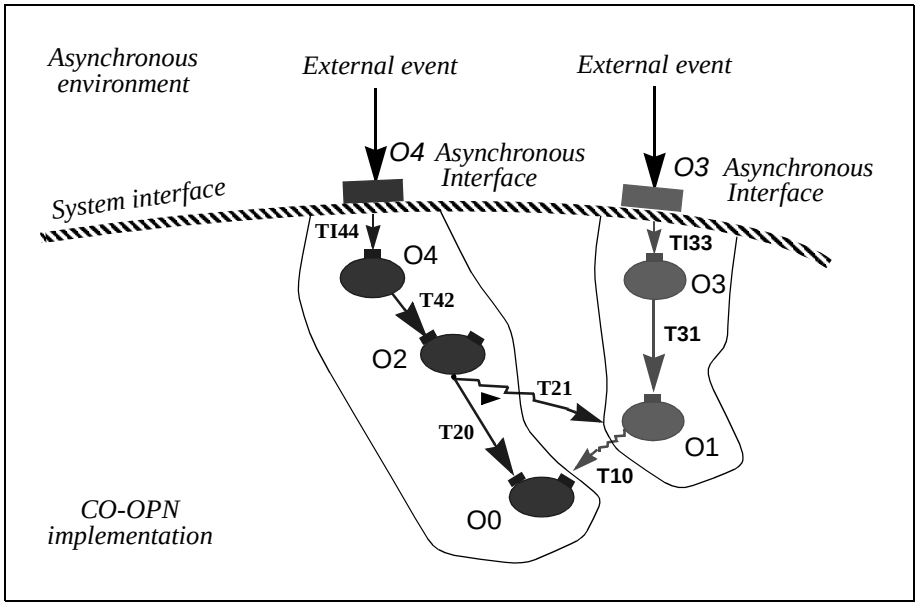


Figure 90. Deadlock Involving Transactions Rooted at `O4` and `O3`

CO-OPN models being purely hierarchical, the opportunities for deadlocks are rather limited compared to systems where general graph dependencies may exist. An example of deadlock is however given in Figure 90, where subtransaction `T21`, which is bound to `T20` by the sequence operator, is waiting for Object `O1` to terminate `T10`, while `T10` is waiting for `O0` to be freed by `T20`. Deadlocks may be handled essentially in three ways:

- They may be *prevented* by holding back transactions until it is certain that the statically predicted set of needed Objects is really available. This would eliminate nearly all possibility for concurrency in our implementations, since transactions tend to spread all over the network as a consequence of the stabilization process.
- They may be *avoided* by temporarily aborting all transactions which might represent a danger.
- The third solution is to *detect* deadlocks when they are already formed, and breaking them by aborting as few transactions as possible. This requires exchanging information with other nodes once a deadlock is suspected in order to acquire a more global view of the situation.

We have chosen the deadlock avoidance strategy, in particular the *Wound-Wait* method [Rosenkrantz et al 78], because it is simple to implement and fits well with the rest of the design. The principle is to assign a global priority criterion to each transaction: This criterion is to compare the respective ages of the top-level transactions. The age is based on timestamping [Lamport 78], but in order to ensure a total order, another global relationship, e.g. the priority of the top-level Object, must be used in order to decide between equally timestamped transactions.

The *Wound-Wait* method states that older transactions must always have higher priority for locking an Object. If a younger transaction already locks an Object, it must be aborted (locally) and retried later. After a finite time this younger transaction will become comparatively old enough to be given maximal priority throughout the whole system.

```
IF timestamp(T2) < timestamp(T1) THEN
  put_back(T1)                          "wound"
ELSE
  halt(T1)                              "wait"
END IF
```

Figure 91. The Wound-Wait Method

The call `put_back(T1)` says that all inferiors of `T1` must be aborted, while `T1` itself is undone and put in a list of pending requests, coming from independent transactions, and from which it must be explicitely extracted and reactivated when `T2` has terminated.

6.3.3 Detection of Stability and Termination

The synchronous/asynchronous interfaces constitute privileged access points for querying about the state of the distributed CO-OPN implementation, since all activity enters the system through these points. Information such as stability and termination may be easily obtained there.

The notions of stability and termination are closely related in CO-OPN: Stability is a necessary condition for the termination, be it of an individual event or of the whole system. The communication model is blocking and the objects are passive, meaning that no activity can appear spontaneously (unless the network is unreliable). In consequence, if any event is currently being treated by the system then one of the entry points must be waiting for the corresponding reply. The following definitions give a gradation in the levels of stability and termination:

- An Object in the system is *weakly stable* unless it is currently involved in the treatement of an event - possibly waiting for a reply - or it has pending events, the treatment of which is imminent.
- An Object is *strongly stable* if it is weakly stable and its last successful operation was to complete an atomic commitment protocol.
- A CO-OPN implementation is *ready to terminate* if all its entry points answer *strongly stable* to an atomic query.

In practice we are only interested in the property of *strong stability* of an Object, since the *weak stability* is a temporary and local condition.

6.3.4 The Synchrony Hypothesis in an Asynchronous Environment

6.3.4.1 The Optimisitic Approach to Simultaneity

The simultaneity operator of CO-OPN specifies the parallel composition of method calls. If some parallel activities involve shared Objects, then it will be necessary to transmit explicitly the fact that these activities are bound by the constraint of strong concurrency. If this information is missing, then the shared Objects will simply treat the incoming requests in sequence because of the natural asynchrony of distributed systems, and thus violate the semantics of the language.

Let us examine the situation of figure 92: Object `O0` is shared by two requests corresponding to the subtransactions `T30` and `T10`. Seeing the given configuration of Objects, it is perfectly plausible to have `T30` arriving at `O0` when `T10` and even `T40` are already finished. In order to ensure the provable properties of the execution, it is then necessary to execute the request of `T30` as if it had happened exactly at the sime time as `T10`, and in any case before `T40`. Since `O0` must satisfy maximally all invocations, it must then somehow return another answer for `T10` and `T40`, so that all requests succeed from its own point of view.

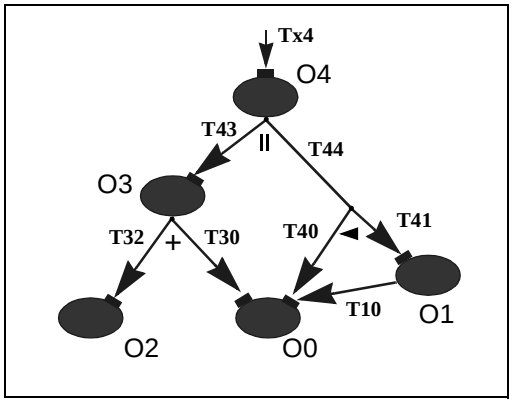


Figure 92. Object Sharing in a Simultaneous Synchronization

There are therefore two problems to answer:

1. How can we tell Object `O0` which invocations are simultaneous, which are sequential, and which are alternatives ? Because of the highly non-deterministic nature of CO-OPN, not even the top-level Object `O4` may know in advance the dynamic configuration of the invocation graph.
2. The shared Objects must be able to provide several answers to the same invocation: This is not a problem, if the messages are logged so that invocations may be reexecuted. Before that, the Object must however be able to return to the state it was supposed to have at the moment of the simultaneous invocations and undo all the intermediate operations and side-effects. Another difficult problem is that the caller (here `O4`) must be disposed to receive new answers for arbitrarily old invocations. That also requires the ability to rollback to the state it had when it was waiting for the corresponding answer. This latter problem is treated within the more general frame of distributed backtracking.

These two problems are discussed separately in the two next sub-sections (6.3.4.2 and 6.3.4.3).

6.3.4.2 The Notion of Synchronization Vector

We have already briefly mentioned Lamport's algorithm for maintaining logical clocks in distributed Objects [Lamport 78]. Its purpose is to provide a view of time which encompasses the causality between events in a distributed system, the most relevant events at that level being usually the sending and reception of messages. For instance, the logical clock of an Object `O0` which receives a message must always be higher than the logical clock of the Object `O1` from which the message was sent. To ensure this property, each message *m* contains a timestamp *TS*(*m*) which is the value of the sender's logical clock. This timestamp

is used by the receiver to adjust its own logical clock: That way it is guaranteed that the logical clocks reflect the causality between the sending and the reception of the message. The following rules are employed, *LC* being the current logical clock and $LC(e_i)$ being the logical clock associated to the event e_i:

$$LC(e_i) := \begin{cases} LC + 1 & \text{if } e_i \text{ is an internal or send event} \\ \max(LC, TS(m)) + 1 & \text{if } e_i = receive(m) \end{cases}$$

Figure 93. Update Rules for the Logical Clock

The next figure shows, based on the configuration of figure 92, the evolution of logical clocks with plain Lamport update rules (we do not increment the logical clock of `O4` between the sending of the simultaneous synchronization messages to `O3` and to `O0`).

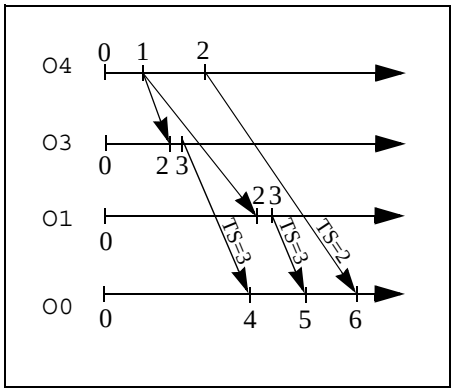


Figure 94. Logical Clocks and Simultaneity

We can see that Object `O0` never receives messages with timestamps appropriate for indicating the required synchronization: The request `T40` is to happen after `T10` but has a lower timestamp than its intended predecessor. Lamport's logical clocks have the property that if an event e_1 causally precedes event e_2, and this may be noted $e_1 \overset{c}{\Rightarrow} e_2$, then its associated logical clock will be inferior (the relation $\overset{c}{\Rightarrow}$ is transitive):

$$\textit{Lamport's Clock Condition:} \quad e_1 \overset{c}{\Rightarrow} e_2 \Rightarrow LC(e_1) < LC(e_2)$$

This condition is unfortunately too weak, because, by looking at the logical clocks associated to two events, it is not possible to tell whether they are to happen simultaneously: The timestamps reflect among other things the number of Objects which take part in the synchronization, which is not relevant for our problem. The main problem is that this timestamping scheme establishes a total order between events.

Let us now try a timestamping scheme which is supposed to reflect the instantaneity of all synchronizations, except for sequences.

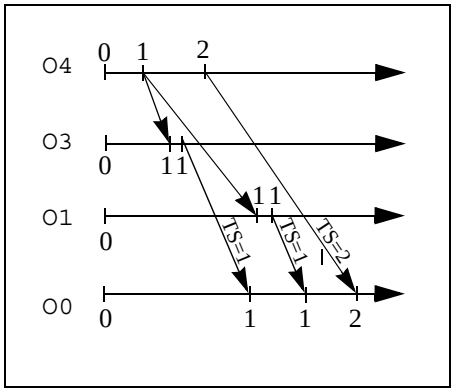


Figure 95. Timestamping with Instantaneous Synchronizations

This figure is closer to our needs since identical timestamps may be used as indication for simultaneity. This scheme works as long as there are no sequences nested within a simultaneity: Figure 95 is ambiguous about the interpretation to be given to the last method call `T40`. It might be understood as the synchronization `T30`&(`T10`..`T40`), which is correct in this situation. But it could as well have been the synchronization `T30`&`T10`&(`T4x`..`T40`) where `T4x` is another transaction created by `O4`, but not passing through `O0`. In this latter case `O0` would be to serve three simultaneous synchronizations instead of only two: This distinction is crucial in CO-OPN. We also still don't know at this point how to cleanly handle the complex synchronization expression at `O4`. The issue is that we really need a way to reflect the hierarchy of synchronizations.

Let us briefly examine the notion of *vector clock* (see [Schwartz&Mattern 92] for a survey) which has the property of preserving partial order between events. It consists of a vector of logical clocks maintained at each site and updated with the ones received as timestamps, in a manner similar to Lamport's simple clocks. Each entry of the vector consists of a logical clock assigned to a specific Object. If we adapt vector clocks to take into account the fact that synchronizations (apart from the tail of a sequence) are considered instantaneous, and if we add entries for representing the operators of compound synchronizations, we obtain the follwing figure:

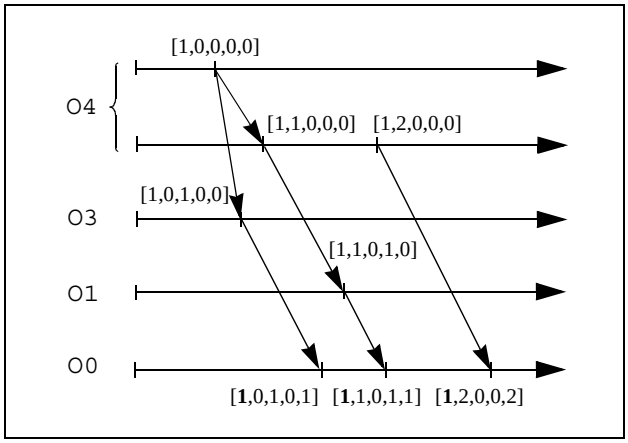


Figure 96. Vector Clocks and Simultaneity

This time, `O0` can see, by examining the entry corresponding to Object `O4` in its vector clock (the numbers in bold face), that a simultaneity is requested. By comparison of vectors, it is also possible to determine where the sequential synchronizations are to occur. This solution is of course very static since it bounds the complexity of the synchronization expressions. We will therefore from now on collect only the information which is strictly necessary, by piggybacking it in parallel with the subtransaction Xids which are created for each level of synchronization. By using additionally the lock histories of the Xids, we can now distinguish between simple (`basic`) and simultaneous synchronizations: The latter correspond to the entries which have identical Xids, such as all the entries with the Xid `Tx4` below.

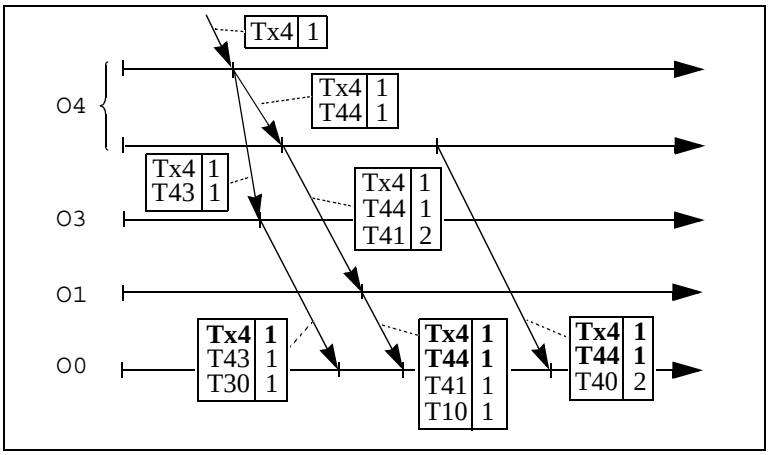


Figure 97. Combination of Xids and Clock Vectors (without Stabilization)

Our problem of determining the simultaneity requirement is related the issue of guaranteeing *causal broadcasts* [Birman, Schiper&Stephenson 91], for which vector clocks are

also used. A typical example where causal broadcast would be useful is on Usenet in the News facility: When somebody broadcasts a question to the net, it often happens that other users read an answer to that question before seeing the question itself. With causal broadcasts, this problem would disappear since the answer would be hidden to each user until the reception of the corresponding question(8). In our case, the highly non-deterministic nature of CO-OPN implies that any communication may be aborted as a result of backtracking and therefore there is no use for an Object to wait for a moment where it would be sure that it has received all the branches of a simultaneity operator: No synchronization is definitive. In compensation, the ability to undo all operations allows the Object to return at any moment to the point where all simultaneous invocations were supposed to take place.

Unfortunately the information provided by the vector clocks is still not sufficient, because we need to distinguish simultaneity from alternatives: In both cases the logical clock associated to the sending event must be the same, even if alternatives are evaluated serially. Therefore we will introduce some symbolic information which, for each message sent, tells exactly the kind of synchronization expected: `sim`, `seq`, `alt`, `basic` or `stab`(9). This data can be transferred in the form of a small enumerated type.

For the `basic` value, the logical clock is no longer necessary. For the `sim` synchronization, we have to add a numeral value which tells the priority to be given to the different branches of the simultaneity: We will explain later how this priority is fixed and exploited. For the `alt` synchronization there is also a numeral value which assigns a kind of priority: This is only a hint which conveys a preference given to the different branches by the emitter of the synchronization. This preference is needed by Objects receiving several of these alternatives in parallel because it is not possible in the general case for a single Object to evaluate more than one alternative at a time: It must therefore serialize the requests in the order given by the assigned preferences.

Within `seq` and `stab` synchronizations we are only concerned about the relative time differences between the various sub-branches of the synchronization: Therefore we will instead consider the timestamps as a way of numbering the subtransaction branches. This numbering is not strictly useful in the cases where sequences are performed serially, since the physical time then clearly tells which invocation comes before the other. We will however for the moment keep the numbering for the sake of clarity(10).

8. In technically correct terms, we would say that the message with the answer has been *received*, but is not *delivered* before the message conveying the associated question.

9. `Stab` is not a kind of synchronization available at the specification language level, but it is needed in the implementation.

10. The numbering might also be used as timestamp indicating the intended order of events in cases where sequential synchronizations are parallelized (optimization not considered in this report because of the high probability of conflicts this strategy generates).

The numeral values associated to `seq`, `sim` and `alt` may be restrained to an arbitrarily small range since they number the branches of a synchronization at the specification level: The compiler may for instance disallow specifications with more than 256 operands for any simultaneity operator. By implementing these numbers as single bytes, we achieve some space saving, both in memory and particularly in the messages. The `stab` value, on the other hand, will need more space for its associated numeral value, since the number of stabilization steps, although finite by hypothesis, is determined purely dynamically.

Definition 41: *Synchronization Vector*

The *synchronization vector* of a transaction *t* is the stack of synchronization operators (`sim`, `seq`, `alt`, `basic`) and stabilization requests (`stab`), with their associated numeral values, needed for any Object participating in *t* to calculate the context of the method calls encapsulated by *t*. Synchronization vectors are only propagated downwards, i.e. in the sense of the directed acyclic Object dependency graph. At any moment the synchronization vector has the same length as the Xid it is associated to and there is a one to one correspondence between their entries.

◊

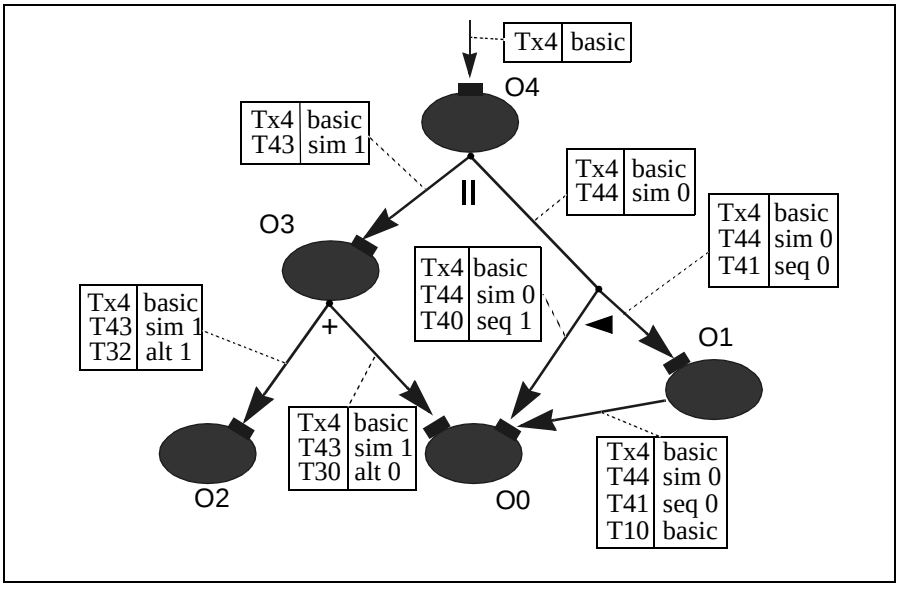


Figure 98. Xids and Synchronization Vectors (without Stabilization)

The difference between vector clocks and synchronization vectors is that vector clocks usually have a static size equal to the number of objects of the system (or a fixed subset of it)[(11)], whereas synchronization vectors grow dynamically and their size is bounded by a factor of the largest possible synchronization in the static Object dependency graph.

The following figure shows that the shared Object O0 always receives the same synchronization information, whatever the delays or local computations times at the superiors. The parts in bold face are in each case common to at least two invocations: Object O0 proceeds by comparison of Xids and synchronization vectors for determining exactly how it is supposed to behave.

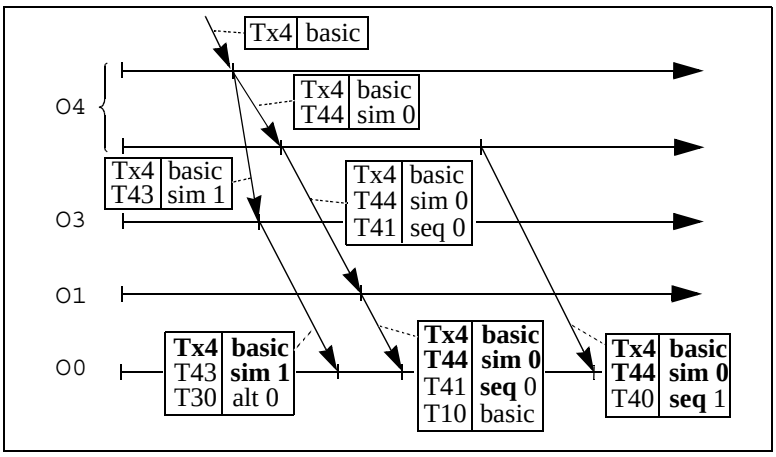


Figure 99. Synchronization Vectors and Simultaneity (Without Stabilization)

We have now all the data necessary for shared Objects to reconstitute internally the synchronization tree above themselves[12]. We will see below how this information is exploited.

6.3.4.3 How to Compute the Context of an Invocation

We have seen that shared Objects must be able to provide several answers to the same invocation: This is not a problem, since all the messages are logged so that invocations may be reexecuted. But before that, the Object must however be able to return to the state it was supposed to have at the moment of the simultaneous invocations and undo all the intermediate operations. We already know that Objects must save their state at the beginning of each subtransaction, i.e. at least before each method call, be it received or emitted. Figure 120 on page 234, which corresponds to Object O0 in the situation of figure 98 where T10 is terminated, T40 running, and T30 not yet arrived, gives an idea of what the internal data structures could look like.

11. Some optimizations are possible to reduce the size of the messages, at the price of additional storage or computing [Schwartz&Mattern 92].

12. We use the term *tree* intentionally even in the cases with Object sharing, because the shared Objects are to be considered as being *split*, as implied by the semantics of rule BEH-SIM. If an Object is shared within a sequence or a stabilization, then it is only the superposition of the trees corresponding to different moments of the execution which will make the synchronization tree look like a graph.

How to Calculate the Context of a Simultaneous Synchronization

Let us demonstrate how an Object determines the context of an invocation, i.e. the state it is supposed to restore whenever a simultaneous invocation arrives out of order. The state is expressed in terms of markings, since the basic model is the Petri net. The inference rules indicate precisely how the marking evolves as synchronizations proceed. Let us define some simple functions for determining the marking at any given moment, by assuming that each completed event makes available the set of consumed and produced tokens. Rule MONOTONICITY allows us to place the firing of an event e within any context m_0. We may thus recalculate from any initial state m_0 the state m_1 resulting from the firing of e by knowing its multi-set pre_e of consumed tokens and its multi-set $post_e$ of produced tokens:

$$\text{MONOTONICITY:} \quad pre_e \xrightarrow{e} post_e \Rightarrow m_0 + pre_e \xrightarrow{e} m_0 + post_e$$

The latter expression is equivalent to $m_0 \xrightarrow{e} m_0 - pre_e + post_e$, hence the resulting state we are looking for is $m_1 = m_0 - pre_e + post_e$. This result is well known in the field of Petri nets [Reisig 85]. If event e is a compound expression with a synchronization as in CO-OPN, then it is useful to be able to calculate the state $StateAt(e)$ at the beginning of e: This is simply $StateAt(e) = m_0 - pre_e$ since the tokens of the precondition of e have already been consumed, whereas the tokens of the postcondition of e have not yet been produced.

Let us apply this relation to the case where $e = e_1 \& e_2$ by combining rule MONOTONICITY with BEH-SIM. If event e_2 occurs after e_1 (due to the asynchrony of the implementation), it is possible to calculate the state in which e_2 should really start evaluating its precondition:

$$StateAt(e_1 \& e_2) + pre_{e_2} = (m_0 - (pre_{e_1} + pre_{e_2})) + pre_{e_2} = m_0 - pre_{e_1}$$

This confirms that both e_1 and e_2 fetch their resources at the same time from a common initial state m_0. From an operational point of view we can consider that two simultaneous transition systems TS_1 and TS_2 must be evaluated in the following order: $pre(TS_1)$, $pre(TS_2)$, $event(TS_1)$, $event(TS_2)$, $post(TS_1)$ and finally $post(TS_2)$. Each of the couples $op(TS_1)$ and $op(TS_2)$ with $op \in \{pre, event, post\}$ are commutative since the simultaneity operator is itself commutative. Concerning the sequence operator, transition systems $TS_1 \,..\, TS_2$ must trivially be evaluated in the following order: $pre(TS_1)$, $event(TS_1)$, $post(TS_1)$, $pre(TS_2)$, $event(TS_2)$ and finally $post(TS_2)$. The same order is valid for the evaluation of two consecutive transition systems during a stabilization.

In the previous we assumed that all pre_e and $post_e$ were readily available. In fact we will need two functions for calculating each of these multi-sets:

```
Consumed _ _ : marking, sync -> marking;
Produced _ _ : marking, sync -> marking;
```

These functions take as input an initial state and return the set of tokens consumed, respectively produced, by the given synchronization. When *e* is a simple method call or transition we can obtain by a low-level mechanism the multi-sets of consumed and produced tokens[13]. We call the respective functions `Pre` and `Post`.

Returning to our example of figure 98, let us suppose that the invocation encapsulated in `T30` arrives at Object `O0`. The request corresponding to `T40` is then abandoned, because it is to happen after `T30`, and if its execution is finished and the reply already sent to `O4`, it will need to be restarted by sending to Object `O4` the message `RestartDependent(T41)` [14]. The tokens originally taken by `T40` are now available for `T30`, which will be executed within the Object state defined by `ObjBefore(T43) - Consumed(T10)`, using the terminology of appendix A.4.

6.3.5 The Global Stabilization Process

In this section we are interested in the way a set of Objects coordinates in order to ensure that a maximal degree of stability has been achieved. The stabilization steps which are performed internally to each Object will be treated later in the report. For the moment we will only investigate the exchange of information which is necessary for a top-level Object to guarantee that the current event has produced all the effects it was supposed to before the atomic commitment protocol can start.

6.3.5.1 Identifying Stabilization Requests

According to the inference rules of CO-OPN, there are two moments where a set of Objects must be stabilized: after each synchronization (rule BEH-SYNC) and in the middle of each sequence (rule BEH-SEQ). We have seen in the previous sections how subtransactions are identified by Xids and how this information is completed with synchronization vectors: This allows Objects to precisely situate incoming requests in their respective contexts. For the stabilization we will need additional communications to take place. The new kinds of messages are the command `stabilize` and the reply `stabilized`. The stabilization itself always succeeds if it terminates. The role of the reply is simply to tell when the stabilization is over, because a synchronization is not considered as successful before its subsequent stabilization is finished. We could have chosen to let Objects start their stabilization automatically after each method call, without having to wait for the command `stabilize`. We preferred however to control explicitely the stabilization process so that it is easier to undo small parts of it when necessary. Another reason is that we need a way to tell an Object

13. This information is collected at the end of each method and transition execution by traversing the associated stack of *undoable operations*. This stack is needed for supporting the backtracking underlying the resolution process. The `Consumed` and `Produced` tokens are stored in the internal structure as shown in figure 120 on page 234.

14. See appendix B, page 237, for an explanation for this message.

shared by simultaneous invocations when it is allowed to merge again and to stabilize the reconstituted Object: This may take place when the shared Object receives its first stabilize request with a synchronization vector where the corresponding `sim` information has disappeared (compare step 9 with all other requests received by Object `O0` in figure 100 below). Still another reason to separate the method call from its stabilization is that its reply may be immediately checked by the caller: Stabilization is then requested only if the reply conforms with the local conditions at the caller, otherwise a `retry` message will be emitted, and no stabilization will have to be undone.

Each `stabilize` command is encapsulated in a subtransaction identified by a Xid. This allows the stabilization to be committed or aborted, like any other request. The reply `stabilized` indicates the `stabilize` command it is associated to by returning this Xid.

It is primordial to transmit a complete view of the current synchronization for the stabilization to be correctly processed. The follwing figure illustrates well that point.

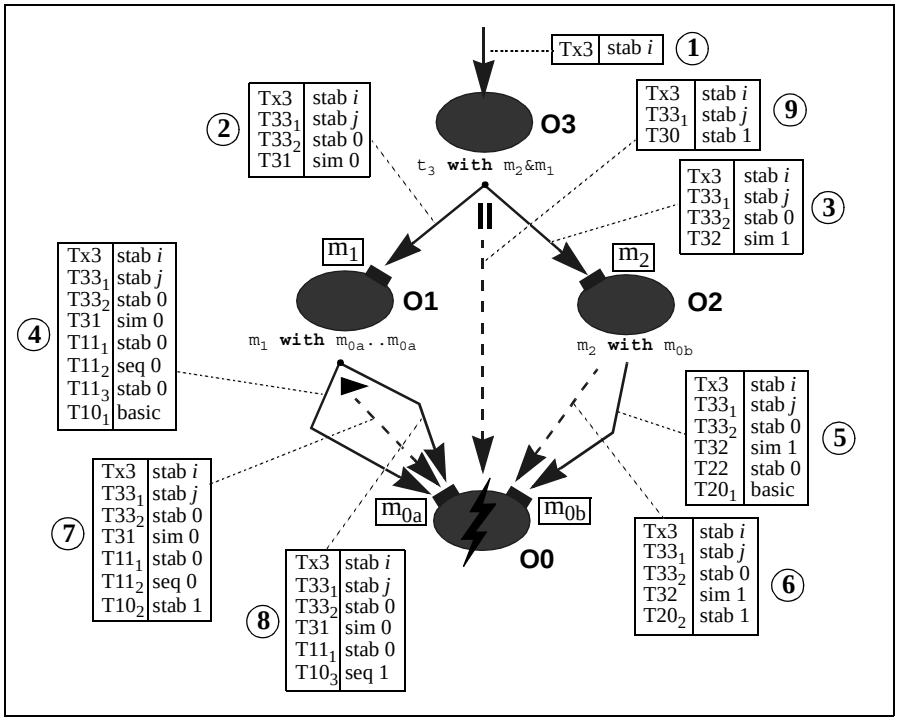


Figure 100. Structure of a Synchronization with Some Stabilization Requests

Figure 100 displays a possible ordering of steps for the given synchronization: Only successful invocations are visible here, and, in order not to overload the picture, the corresponding replies are not visible, and neither are the stabilization steps between (8) and (9). This synchronization is nested inside the stabilization of Object `O3`, and this fact is directly

discernible in the different Xids and synchronization vectors. The numbers *i* and *j* accompanying the `stab` entries of `Tx3` and $\texttt{T33}_1$ indicate that Object `O3` is the i^{th} Object to be stabilized at this stage and that the synchronization we see here is the j^{th} stabilization step of Object `O3`, i.e. the j^{th} transition it executes. In general, a `stab 0` is a place holder for a stabilization which has not yet begun.

Recall that in CO-OPN, when an Object `O0` receives two simultaneous invocations ($\texttt{T10}_1$ and $\texttt{T20}_1$) it will appear as being split into two independent sub-Objects: This is symbolized by the crack in `O0`. In step number 7, which is in the middle of a sequence, itself in the middle of a simultaneity, a stabilization is requested: The structure containing the associated Xid and synchronization vector tells Object `O0` that the stabilization is to be performed on its left sub-Object, i.e. the part which served the first call to method $\texttt{m}_{0a}$ (step 4). The dashed arrows correspond to `stabilize` messages; to be really accurate we could have made them point to methods called `stabilize`.

6.3.5.2 Organization and Optimization of The Stabilization Process

As we have seen in chapter 3, the notion of stability is not easy to capture since it may entail new synchronizations which themselves will need their own stabilization. The language designers' basic idea was to produce a maximal reaction to any external event. From the implementation point of view, an Object is stable when it has ensured that no more of its internal transitions are firable. This condition may depend on the state of other Objects when a transition wants to synchronize with an external a method: If the local state of the stabilizing Object allows this synchronization to take place, but not the state of the other Objects, then the synchronization is postponed and the stabilizing Object is considered as stable until the state of the other Objects makes the synchronization possible. This is the anti-inheritance of instability that we presented in section 3.6.7.

In the semantics of CO-OPN, the computation of all the possible behaviours is achieved by starting from the lowest Object in the hierarchy and gradually incorporating new Objects from above. At each stage a state graph is constructed, as in example 2 on page 69. This allows us to determine, if several Objects become simultaneously unstable, that the lowest among them has the highest priority in the case where they compete for a limited set of resources. The semantics does however not tell how these Objects instantly learn that there is a possibility for them to make one further step of stabilization.

The following example (figure 101) is similar to the one in section 3.6.7 on page 77. Let us suppose that Object `O2`, which was stabilized in the middle of the sequence rooted at `O4`, suddenly becomes unstable again, because the state of `O0` has changed after the invocation of $\texttt{m}_{0b}$, making its method $\texttt{m}_{0a}$ firable. Object `O2` may thus fire its internal transition $\texttt{t}_2$ and become stable again. The problem, from an operational point of view, is to determine how `O2` may learn that method $\texttt{m}_{0a}$ has become firable.

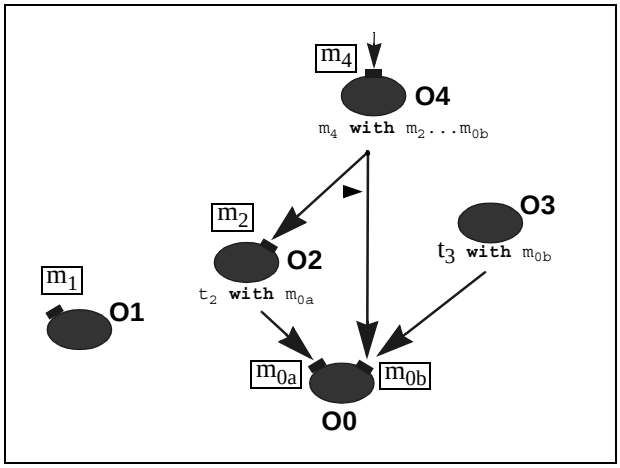


Figure 101. Example where Stabilization of o2 is Reactivated after the Call from o4 to o0

There are three possibilities:

1. Object o2 continuously keeps calling method m_{0a} until a positive answer is received. This solution has the usual drawbacks of busy waiting, the cost of which is amplified in a distributed environment.
2. Object o0 remembers that o2 failed a stabilization step on method m_{0a} and sends him a message to tell that method m_{0a} has become firable. The problem here is that it breaks the general hierarchical design by making an Object (o0) reponsable for Objects (o2) which are above itself in the hierarchy. Object o0 may well overlook[15] firable events in o3 or o1 because it does not have a complete view of the situation above itself.
3. Object o4 sends a message to every Object below itself, following the total order $\sqsubset$, telling them to stabilize again, even if their own state has not changed. In this technique, the resources are exploited in a more rational way than in proposition 1, the hierarchy is respected and completeness is ensured, by opposition to proposition 2. The inconvenience is that by following blindly the total order, even completely unrelated Objects (here o1) are contacted, thus slowing down the whole process, since these other Objects may well be engaged in independent transactions.

We propose a combination of the latter two solutions. The goal is to respect the hierarchy, to fire exhaustively the firable events, and to try to restrict the stabilization process to a set of

15. The firing of an internal transition is said to be *overlooked* if the Object it belongs to accepts a method call before the transition is fired. This violates the semantics of CO-OPN because the Object would then be unstable when serving the method call.

relevant Objects. Only the Objects the state of which has changed (O0), and the Objects which depend directly on the latter for their stabilization (O2 and O3) will be considered.

- The Objects the state of which has changed are simply the Objects which have successfully replied to a subtransaction. The list of Xids referring to these Objects is automatically transmitted along with the commitments (see *Locking Rule* #ii on page 153).
- The list of Objects which depend directly on the previous group for their stabilization may be consituted simply by a static knowledge of the configuration of the hierarchy. We suppose that everyone has a copy of a table *StabClients* which gives the list of Objects depending on a given other Object for their stabilization: *StabClients*(O0) would return the names O2 and O3. Another dynamic and more accurate solution would be to exploit the idea of proposition number 2 above, i.e. to transmit as additional argument of a committed subtransaction the entire list of Objects which have failed a stabilization step on the modified Objects and which will have to be reactivated by O4 to check whether the failed synchronization can succeed now. In our example, the reply of method m_{0b} would return the names O2 and O3 as additional information. Unfortunately, in order to be more accurate than in the static solution, it is necessary for the couples O2-O0 and O3-O0 to continually tell each other whether their local states still allow the synchronization to take place. If this is not done, then the subtransaction reply messages would just return a full copy of the relations which we proposed to store in the static tables. Therefore we choose the table solution.

The following figure describes how the entire synchronization and stabilization take place. We can see that Object O1 is effectively excluded, although its position number would logically make it participate in the stabilization, since comprised between O0 and O2 which themselves take part in the process.

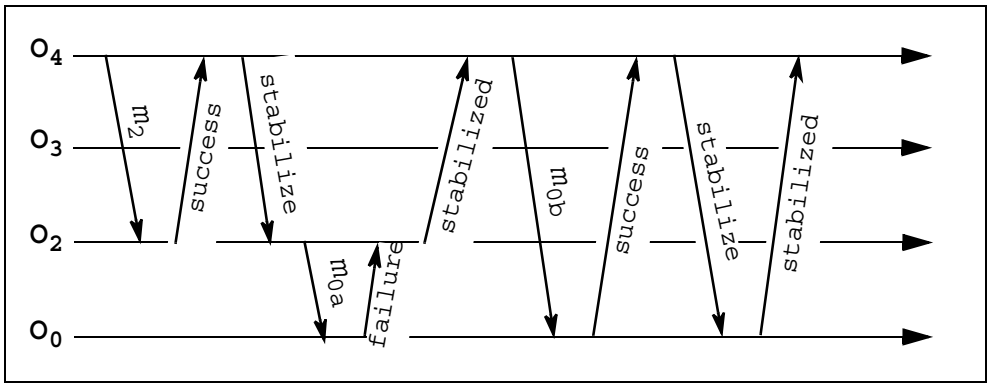


Figure 102. Synchronization and Stabilization for figure 101 (1st part)

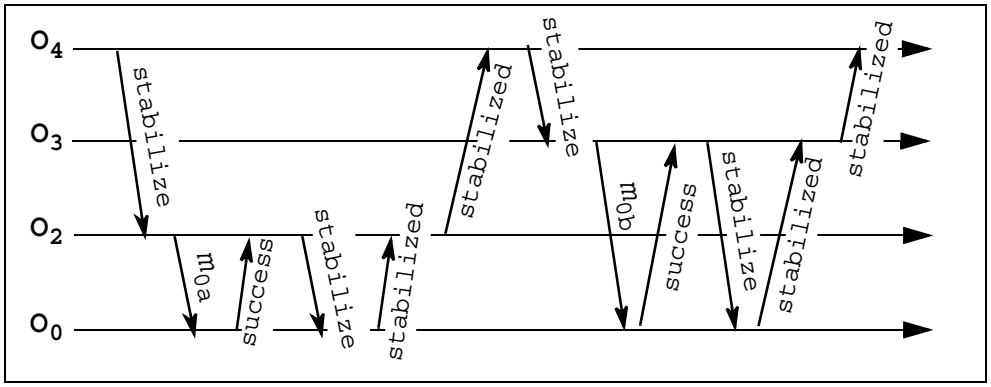


Figure 103. Synchronization and Stabilization for figure 101 (2nd part)

In the examples described until now, all objects were connected rather tightly, by direct client-furnisher relationships. We have yet to verify that our method also works in the cases where Object hierarchies are loosely connected, like for instance in the following figure, where the activity advances in three phases from `O5` to `O4`. The dashed lines represent `stabilize` commands.

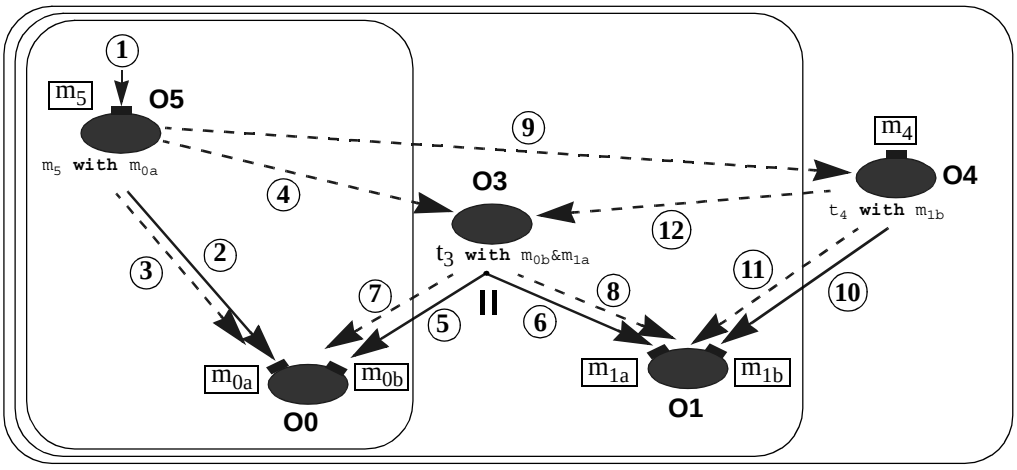


Figure 104. Stabilization by Extending Spheres

The hierarchical stabilization algorithm (function `StabilizeLowerObjects`), given in appendix A.3, maintains a set *ToStabilize* of Objects remaining to be stabilized. At each iteration, the set *ToStabilize* is augmented with the set of Objects which are *StabClients* of the Objects modified during the previous iteration. This is how the set of “interesting” Objects is extended until there are no more new *StabClients* to take into account. This set may finally include all the Objects of the specification in the worst case. At the same time, the current Object, called *LowestObject*, is always the lowest Object extractable from the set

ToStabilize: This guarantees that among all the Objects having firable internal transitions, it is always the lowest in the set which is chosen, and thus it is ensured that no event is being overlooked by the algorithm. The *LowestObject* may be removed from the set *ToStabilize* if it still is the lowest Object of the set at the end of the iteration. If all partial stabilizations terminate, then the set *ToStabilize* will grow down to the empty set after a finite time.

If the prototyping tool had the freedom to establish itself the total order among Objects, then it would be judicious to assign the highest numbers to Objects like `O3` above, which offer no methods and only serve as auxiliary connectors between Object hierarchies. The consequence would be to confine the transaction as long as possible to the local set of Objects which are tightly connected. By preventing transactions from spreading too fast, locking of new Objects is delayed, with the benefit that some concurrency may be preserved in the system. Our algorithm works even without this optimization and is at least clever enough not to follow blindly the total order, which would inevitably lead to locking every object of the system for each transaction.

The example of appendix E starting on page 267 shows a complete stabilization in the case of the collaborative diary application. It also elicits an unexpected consequence of the total order, which is that the Object at the interface of the system (`GIL2`) cannot perform in an atomic sequence the operations of adding a meeting to the diary and of checking whether it creates a conflict. This is because the detection of the conflict requires the data to transit through Objects `DSA2` and `DSA1`, which are higher in the hierarchy than `GIL2` and which therefore will not be stabilized before the end of the atomic sequence. Consequently, the operation of checking for possible conflicts will have to be performed later, when the whole system has been stabilized[16]. In other words, it is not a perfect solution either to put the auxiliary connector Objects at the highest ranks in the system, as recommended in the previous paragraph.

6.3.5.3 Finalizing the Synchronization and Stabilization

In the previous subsections was described the general case where a stabilization happens during the evaluation of an event. We still have to show how the top-level Object terminates the treatment of an event.

The top-level Object is always activated by an external event transmitted by the corresponding synchronous/asynchronous interface. The property which distinguishes a top-level Object from other Objects is that they must adequately finalize each event. This results in two additional duties, which are to stabilize all the Objects which may be indirectly modified by the current event and to coordinate the atomic commitment protocol. We are interested in solving the first issue, the second consisting simply in executing the standard

16. Here we see the benefit of the call-back facility at the system interface, which serves as a substitute for the stabilization mechanism which is employed within the CO-OPN implementation.

two phase commitment protocol on the set of Objects which have taken part in the transaction.

The top-level Object of a transaction is not necessarily the highest Object in the specification regarding the total order. In other words, because of the anti-inheritance of instability, the effects of a given event may spread upwards and above the top-level Object which the event is rooted at. This means that the top-level Object will have to send `stabilize` messages to Objects which are higher in the hierarchy. For this we introduce a meta-mechanism, which is not part of the normal execution, and which consists in a cooperation between the different synchronous/asynchronous interfaces. This time it is not too embarrassing to invert the client-furnisher relation, because it happens outside of the regular execution scheme. We will always start transaction Xids and synchronization vectors with a special-purpose root so that Objects may know which event the stabilization requests are associated to. If this is not done, the Objects which are already locked will not recognize the commonality and thus consider it as an independent activity which must be set to wait.

An example of execution is given in appendix E. Notice how important the `stabilize` messages are for the event to be propagated as far as possible.

6.3.6 Distributed Prototype Startup

At startup, there are two tasks to fulfill. The first is to ensure that all Objects have a unique number which respects the dependency graph. The second is to coordinate the stabilization needed by Objects which have unstable initial states.

6.3.6.1 Establishment of a Total Order

The assignment of unique numbers which respect the dependency graph can be done either at compile time, or at link time, or at program startup. The last solution requires the election of a coordinator which explores the prototype in order to discover its topology and which establishes the numbering and transmits the assignments to all Objects by the means of special-purpose messages. We choose for the moment a more static solution, which is to generate at link time some definitions, which are part of the model structural description layer of each Object (see figure 79 on page 139). These definitions establish two tables: The first table gives the correspondence between the name and the assigned number of every Object, and the other table implements the `StabClient` relationship, i.e. for each Object *o* it gives the list of Objects which have an internal transition synchronizing with a method of *o*.

6.3.6.2 Stabilization at Startup

Objects will often receive initial markings which make themselves or a `StabClient` of theirs unstable at startup. In order to coordinate the needed stabilization, the highest Object

in the system will enter its `StabilizeLowerObjects` function (see appendix A.3) with the list of all Objects as `ToStabilize` parameter. Then it will have to apply `Stabilize` on itself and to coordinate an atomic commitment protocol before the system is ready to receive input.

6.4 The Resolution Layer

6.4.1 Solving CO-OPN Events by Resolution

Our objective is to find the most efficient way of executing a prototype according to the inference rules which define the dynamic behaviour in CO-OPN. For this we need a *resolution mechanism*. Resolution works by applying the inference rules backwards until the inital goal, i.e. the event, is reduced to the empty goal, thus proving that the system is able to handle the given event [Padawitz 88]. Similarly, if we refer to a normal Prolog execution [Colmerauer 83], we see that the body of a clause is progressively and entirely reduced, working in the textual order, before it can be concluded that the clause is successful.

Compared to the previous, deductive approach of chapter 3, our purpose is now to take the source state of the system for granted, since, in an implementation, resources cannot simply appear when needed. This means that rule MONOTONICITY will no longer be employed. We will proceed likewise for ordering the operations in a temporally correct order: For instance, it is clear that the preconditions of an axiom should be evaluated before solving the associated synchronization or producing the postconditions. Although not needed from the point of view of pure logic, this is required for the system to have a “natural” behaviour, and for the variable assignments within transitions and methods to be performed correctly w.r.t. the expected data flow, since all parameters now have strong modes.

6.4.2 Parallel and Distributed Prolog Variants

We have already many times referred to the Prolog language for describing the notion of search non-determinism and how it is implemented. In this work we also want to exploit existing results in the fields of parallel and distributed implementations of this language, because there are some analogies with the distributed part of CO-OPN. There are two families of Prolog implementations which we are interested in, namely the *process-based* Prolog extensions and the *and-parallel* implementations of sequential Prolog.

6.4.2.1 Process-Based Prolog Extensions

In the class of *process-based* Prolog dialects [De Bosschere 94] the parallelism is managed explicitely at the programming language level, by addition of primitives for dynamically

creating processes and for communicating and synchronizing. These dialects lend themselves well to distributed implementations since they disallow state sharing. The most relevant works for us are Delta-Prolog [Pereira et al 86] and CS-Prolog [Ferrenczi&Futo 92] since they support distributed backtracking. They are however too restricted for CO-OPN, because they do not have the notion of simultaneity, i.e. all requests are serialized, and therefore they are not faced with the same problems as we have with shared objects.

In [Eliëns 92] a description is made of a Prolog dialect which systematically avoids distributed backtracking by memorizing all results from remote predicate evaluations. This requires huge amounts of memory, and would not work in the case of CO-OPN, since we have to take into account the possibility that previously acquired resources may be requested by Objects with higher priority, and vice versa, i.e. resources which were initially not available may be freed by aborted requests.

6.4.2.2 And-Parallel Implementations of Prolog

The other related class of Prolog implementations is based on *and-parallelism* and aims at transparently parallelizing sequential programs in order to achieve better performance. In and-parallelism the sub-goals constituting a given program clause are evaluated in parallel, although they are clearly sequential in the usual interpretation of Prolog. And-parallelism is interesting for us because of its analogy with the simultaneity operator of CO-OPN: In both cases all parallel branches must succeed, as indicated by the *and* appellation. In and-parallelism, the processes are workers which are continuously fed with new sub-goals and environments in which the sub-goals are to be evaluated. Although most and-parallel implementations of Prolog are designed for multi-processor machines with shared memory, the strategies developed for coping with variable assignment collisions may be reused for resolving resource distribution conflicts within shared Objects in distributed CO-OPN implementations as will be explained below.

The most important problem to resolve in and-parallel Prolog is the detection and correct management of *shared variables* [Hermenegildo&Rossi 93], i.e. variables which are taken as arguments by several sub-goals evaluated in parallel. If such a variable is unbound, i.e. unassigned, at the beginning of an and-parallel execution, then only one sub-goal, the *producer*, must be allowed to assign a value to it, and all other participants, the *consumers*, must conform to the decision of the producer since variables may only be assigned once in Prolog. Consider the following example, where *p* is a predicate defined by the clause

```
p(X,Y) :- a(X), b(Y).
```

Meaning that *p* is satisfied for any assignment of variables *X* and *Y* which also satisfy the sub-goals *a(X)* and *b(Y)*. For a call like *p(1,2)*, the sub-goals *a(X)* and *b(Y)* can safely be executed in parallel since they are independent. The problematic case is the call *p(S,S)* which results in the evaluation of *a(S)* and *b(S)* by replacement of the formal parameters by

the shared variable *S*: If *S* is already bound to some value, then both *a* and *b* are consumers w.r.t. *S* and may be executed in parallel, but if *S* is unbound, then one of *a* and *b* must become the producer while the other is necessarily a consumer, and this introduces a dependency between the sub-goals. To guarantee a correct parallelization, three methods have been proposed in the literature:

- To complete a thorough compile-time analysis of the source text in order to detect all situations in which it is safer to execute goals sequentially because it cannot be determined statically whether two given goals will always be independent or not [Chang, Despain&DeGroot 85]. This approach is very pessimistic and leads to poor performance when compared to the following propositions.
- To generate some simple tests to be verified at run-time and which partially replace the previously mentioned static analysis technique. Two goals are then serialized when the dynamic tests show that they are dependent. This method is called *Restricted And-Parallelism* and allows a broader class of programs to be parallelized [Hermenegildo 86].
- The last solution, which allows the greatest amount of parallelism, is to perform all verifications and process scheduling at run-time [Kumar&Lin 86], [Tebra 87], [Drakos 90]: Consumer processes are restarted if it turns out that they have read or generated a value which is overwritten by a producer process. All dependencies are managed dynamically, which may however lead to excessive overhead when exploiting very fine-grained information within advanced *intelligent backtracking* schemes [Drakos 90].

We are especially interested in [Tebra 87] and [Drakos 90] because they present a so-called *optimistic* or *unrestricted* and-parallelism which is designed to work as well on distributed memory machines(17), and are therefore very close in spirit to our approach for implementing the simultaneity of CO-OPN. The secret of their approach is to remember for each variable the list of processes which have accessed them in order to decide dynamically which one is to be considered as a producer w.r.t. the other processes. This mechanism is easy to retranscribe in CO-OPN because of the natural encapsulation of Objects.

Although there is no producer or consumer as such in a simultaneous synchronization, since the participants in a simultaneity are necessarily independent according to the rules of CO-OPN, we will need to institute a hierarchy among requests in order to ensure that all solutions, i.e. token combinations, are explored exhaustively. If this hierarchy is not enforced, then conflicts may well never terminate, since each request will systematically undo the work of its rivals in order to achieve the best results for itself. This is the subject of the next sub-sections.

17. Tebra describes a scheme where all assignments of shared variables are announced to the other processes by message-passing. Drakos uses a hybrid architecture where the variables are located in shared memory, but processes synchronize by message-passing.

6.4.3 Assignment of Object and Request Priorities

In parallel Prolog, in order to establish the producer-consumer relationship, priorities must be assigned to the different processes: The producer is the one which receives higher priority w.r.t. the consumers of a given variable. If the objective is to generate solutions exactly in the same order as in sequential implementations[18], then the producer will be the leftmost predicate in the source text. Therefore process priorities will simply be assigned according to the textual order by decreasing importance.

In CO-OPN, priorities are to be assigned to the different branches of simultaneous requests. We have seen in chapter 3 that the semantics of an Object is built incrementally by taking into account more and more client Objects by following strictly the increasing order of the static dependency graph. If only one token is available for two competing simultaneous invocations, then it must be given to the call coming from the lowest, i.e. the nearest, Object. The priority of a request is therefore determined by the number of the calling Object, and this information is available in the Xid of the enclosing transaction. This rejoins the Prolog point of view that processes are discriminated by their relative priorities.

We still have to find out how to decide about the respective priorities of several requests coming from the same Object. There are two situations to investigate, the first being when the caller is itself shared by a simultaneity coming from above, as in case (*a*) below, and the second being when branches of a simultaneity involve the same participants and must be discriminated arbitrarily, as in case (*b*) of figure 105.

18. This is a frequent requirement since sequential Prolog programs will often not terminate when evaluated in another order than the usual textual order or when parallelized naively, i.e. without testing that the states of the shared variables effectively allow independent concurrent execution [DeGroot 84].

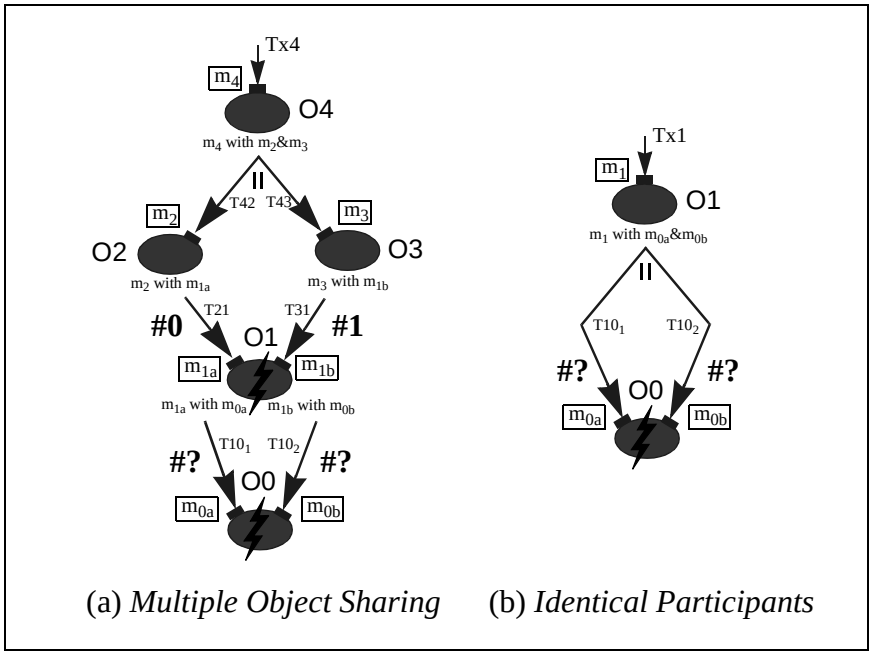


Figure 105. Problematic Priority Assignments in Simultaneous Synchronizations

In the case (*a*) Object `O0` cannot decide which one of $\texttt{T10}_1$ or $\texttt{T10}_2$ is to be given the highest priority if it does not additionally take into account the respective superiors `T21` and `T31`. The priorities of the superiors will therefore be explored upwards by examining the Xids until the first difference is found: In the given situation, $\texttt{T10}_1$ and $\texttt{T10}_2$ receive respectively the priorities of `T21` and `T31`, i.e. 0 and 1. This has the twofold advantage that it is conformant with the semantics of CO-OPN and that the total order is the same for all Objects, in particular for both `O0` and `O1`, which reduces the need for coordination between participants.

In the case (*b*) an arbitrary priority must be enforced, since both requests have the same importance according to the semantics of CO-OPN. We let the emitter of the simultaneity decide, for instance by following the textual order of the specifications, and transmit this information by the means of the synchronization vector, as shown in section 6.3.4. It should be noted that we prefer not to use the serial number of the subtransaction Xids, i.e. 1 for $\texttt{T10}_1$ and 2 for $\texttt{T10}_2$, since these numbers change each time a request is aborted and restarted.

6.4.3.1 The Problem of Deep Backtracking

We have just seen the role of the priorities for shared Objects. It is however important that both the emitter of a simultaneous synchronization and the shared Object(s) have the same view of request priorities, because of the possibility of *deep backtracking* (also sometimes called *outside backtracking*): When the simultaneity is terminated, the flow of control may

proceed forward and a failure may appear in subsequent computation. It is then necessary to (deeply) backtrack into the previously succeeded simultaneity and ask for a new answer.

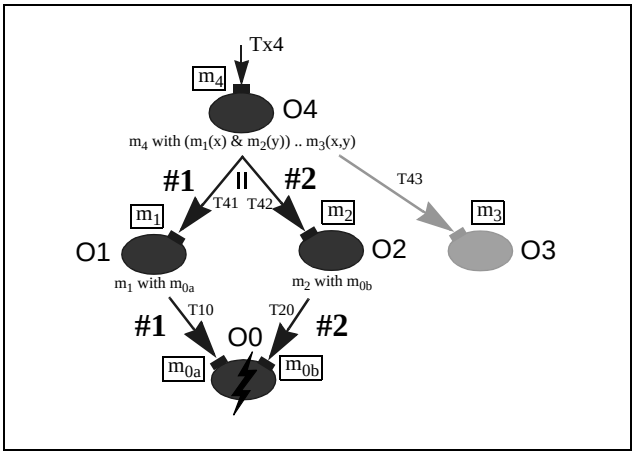


Figure 106. Example of Deep Backtracking from a Sequence back into a Simultaneity

In figure 106, we suppose that the synchronization of `O4` with `O3` fails because one of the parameters *X* or *Y* of m_3 has a bad value. Object `O4` must then send a `retry` message to `O1` or `O2`, which will be forwarded to `O0`, in the hope that this will assign new values to the parameters which are acceptable for `O3`. In order to ensure an exhaustive search for all solutions, `O4` must ask for retrying the request which has the lowest priority w.r.t. the shared Object `O0`. Therefore `O4` must know how the priorities are assigned by `O0` (and by any other shared Object involved in the given simultaneity). We could imagine a scheme where all participants transmit along with the successful replies their own vision of the priorities, so that the emitter may take this into account when sending `retry` commands. This information would however be too local, since the same branch of a simultaneity may have the highest priority at one Object and have the lowest priority for another, as in the next figure:

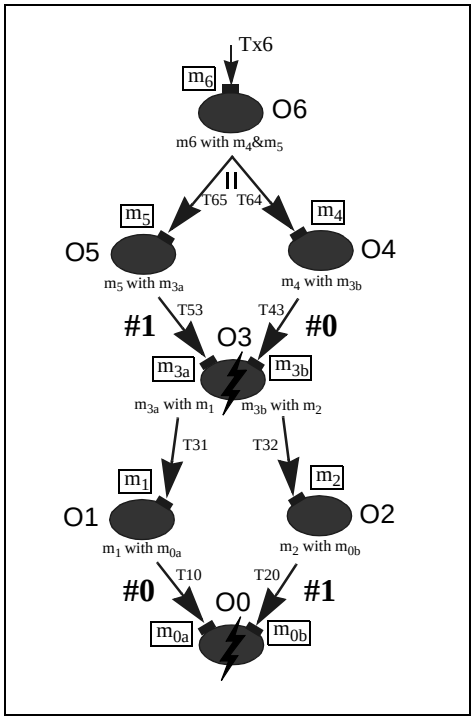


Figure 107. Incompatibility of Request Priorities in Shared Objects

Here we see that the branch `T65-T53-T31-T10` has the highest priority for Object `O3`, while it has the lowest priority for Object `O0`, and vice versa for the branch `T64-T43-T32-T20`. It is therefore not possible for `O6` to establish a globally coherent hierarchy among the branches of the simultaneity it emits.

The solution, since we do not want to exclude multiple Object sharing, is to require that none of the branches of a simultaneity produce such priority inversions. To this end, it is necessary to perform at compile time an exhaustive analysis of all synchronization combinations to verify that emitters and receivers always have a compatible view of invocation priorities. If this condition is not fulfilled, it is up to the developer to renumber the Object hierarchy (in the above example it suffices to permute the numbers of `O4` and `O5`) or reorganize some groups of Objects by merging or splitting them. One frequent situation is the following:

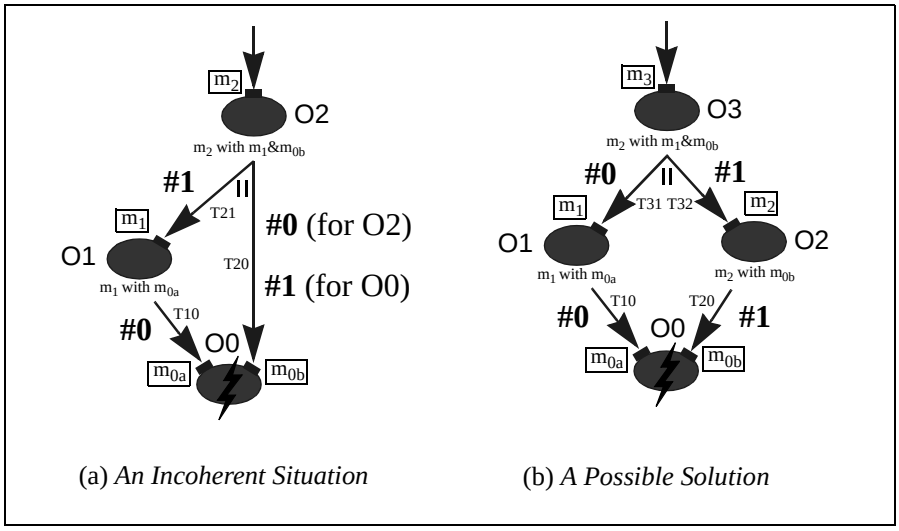


Figure 108. A Disallowed Object Topology and a Possible Remedy

The situation (*a*) of figure 108 is not allowed, because `O2` will assign the highest priority to `T20`, since the target Object `O0` is lower in the hierarchy than the target for branch `T21`, which is `O1`. From the point of view of `O0`, this priority assignment is unfortunately wrong, and therefore we propose instead the topology (b), where the original Object `O2` is split into `O2` and `O3`. Another possibility would have been to merge `O1` with `O0` or `O2`, but in the normal refinement process, as more and more details are added to the specification, Objects tend to grow in size and it may therefore often be more attractive to split than to merge.

6.4.3.2 The Search within Sequences and Stabilizations

All remarks done so far about the distributed resolution mechanisms concerned the simultaneous synchronization operator of CO-OPN. Sequences and stabilizations are straightforward to implement in comparison with the simultaneity because it is possible to implement classical sequential backtracking schemes at all levels without establishing artificial priorities: The chronological ordering defines a natural hierarchy among requests. It is however very difficult to optimize the related distributed backtracking mechanism:

- The cumulated effects of non-deterministic choices made during the search may lead to highly unstable behaviours. The consequence is that the progress in the direction of a solution will often be very irregular, something which makes optimizations difficult. In [Drakos 90] was defined an intelligent backtracking mechanism which could be useful for implementing sequences and stabilizations in CO-OPN, since it allows a form of distributed *dependency directed backtracking* [Stallman&Sussman 77], i.e. backtracking beyond several choice points, which are *not* reset, in order to directly retry the choice point which has been identified as responsible for the current failure in the search process. The pur-

pose in not resetting the intermediate choice points is to avoid undoing computations which are believed to contribute to the currently elaborated solution. These choice points must however be somehow remembered and eventually reset in order to guarantee the completeness of the search. Drakos himself admits that the quantity of dependency information to manage is so high, and that the interaction protocols are so complex, that his clever algorithm is outperformed by the more sequential ones.

- A mechanism which is simpler than dependency directed backtracking is the one called *backjumping* [Gaschnig 79], the principle of which is also to identify and directly backtrack to the choice point responsible for the current failure, but this time the intermediate choice points are reset. This optimization is very common and unexpensive to implement, the problem being solely not to backjump too far and miss potential solutions. This technique may be applied locally to the Object application layer as a step of the incremental prototyping process, as we will show in the next chapter: The developer may thus greatly ameliorate the efficiency in the search for the combination of input tokens which allow firing the current event. Unfortunately the synchronizations may not be optimized that way since there is not enough global information available at the user-level or even at lower layers to ensure that solutions will not be missed by backjumping.

6.4.4 Levels of Parallelism Allowed by the Resolution Layer

To end the presentation of the resolution layer, let us discuss the different levels of inter-Object parallelism which could be practiced in simultaneous synchronizations. In increasing levels, we can mention the following:

1. Completely sequential: The branches of the simultaneity are emitted one by one, each one being activated as soon as the reply for the preceding has been received.
2. Emit all branches in parallel except those which might induce Object sharing. This is similar to the and-parallel strategy of [Chang, Despain&DeGroot 85] where static analysis is used to detect all situations in which it is safer to execute goals sequentially.
3. All branches are emitted in parallel and are serialized by shared Objects. If a request with high priority is received or retried in a shared Object later than a request with low priority, then the latter must be restarted (by sending an adequate `RestartLower` message to the emitter of the simultaneity, see appendix B) in order to conform to the result of the former. A low priority request may need to backtrack because of a conflict with higher priority requests within a shared Object: To prevent solutions from being missed, it must wait for the *right to backtrack*, which is a special message circulating along the branches of a synchronization and which guarantees that all requests with higher priority have attained a dependable solution. In fact the right to backtrack is allotted to a synchronization branch when its sibling with immediately higher priority has terminated. This is the optimistic and-parallel approach of [Tebra 87].

4. Use the same strategy as before, but instead of resorting to a right to backtrack, requests with low priority are systematically restarted by shared Objects when a request with higher priority backtracks, resulting a generalized usage of the message `RestartLower`. This is simpler to implement, generates more parallel activity and should lead to slightly better global performance in programs with little or no non-determinacy, which is one of the main purposes of the incremental prototyping process.
5. Using the same resolution mechanism as above, we weaken the notion of conflict. Until now we had a strict ordering of tokens established at the beginning of a simultaneity, in order to guarantee the completeness of the search. The highest priority request had to first try the tokens with lowest indices and so on. Instead of this, we can allow the ordering to be dynamically redefined each time a higher priority requests arrives: The advantage of this scheme is that lower priority requests are not obligatorily restarted. The price to pay for this optimization is a heavier iteration mechanism: It may be necessary at each choice point to remember the complete set of tokens (or their addresses in memory) which have already been tried. If there are few tokens in each place, then the cost will be negligible or even inferior to more classical iteration schemes. If on the other hand the places contain many tokens then the resulting non-determinacy will also be higher, leading anyway to poorer performance because of backtracking. The remedy is to avoid considering places as a general-purpose repository and instead to define the tokens as structured data types for storing the elementar values. That would be a highly recommended refinement step for the specifications. This is why we adopted this solution for the implementation of CO-OPN.
6. Instead of systematically restarting lower priority requests upon conflicts in shared Objects, it is possible to directly send new replies. For this to work properly, the caller must still have an internal state which enables him to accept these new replies and return to the point where he was initially waiting for such a reply. This implies for instance that these requests have not been aborted in between.

This ends our presentation of the control layers of CO-OPN Objects. The next section covers the issues in generating prototypable application layers.

6.5 The Generated Code

6.5.1 Overview of the Model Structural Description Layer

The *model structural description layer* is essentially an auxiliary layer which serves as a binding between the application layer and the run-time support of CO-OPN prototypes. It contains:

- The intra-Object structure: In the form of data structures or functions, a mapping between the places and the transitions in order to help the control layer accelerating the search for the next firable event. This is a classical optimization used even in the field of compilation of Petri nets [Colom, Silva&Villaroel 86], [Taubner 87].
- The inter-Object structure: The structural description layer is a practical place for defining frequently changing tables relative to the topology of the system, as mentioned in sub-section 6.3.6.1. These are data which are orthogonal to the intra-Object functionality and should therefore not be declared at the application level.
- The auxiliary definitions necessary to have convenient interfaces to *send* and *receive* primitives at the application level: This is necessary since we want to hide all the lower-level arguments needed by the underlying control layers. We also need type definitions and functions for saving in the internal control structure (see appendix A.4) the specified parameters for sent and received method calls.

Let us now proceed to the description of the application layer.

6.5.2 The Application Layer

6.5.2.1 General Objectives

The main objective in the design of the application layer is to support our incremental prototyping methodology. This means that we focus more on software engineering principles such as safety, modularity and legibility of the source code than on the resulting execution efficiency, although this latter issue has not been neglected either. To this end, it has been decided that the application layer should be orthogonal to the resolution, concurrency control and fault-tolerance mechanisms. For the latter two points this seems to be well accepted principles. It was however a challenge to design a resolution procedure which allowed executing application code written in an imperative language. A final concern is that we want to promote the independency of the methodology from the target language and tried therefore to limit the use of language-specific constructs.

6.5.2.2 Mixing Procedural and Logic Styles

Since the programming language targeted by OOMP is of the imperative kind, it means that CO-OPN constructs belonging to the logic paradigm will need some adaptation to be implemented. This concerns the use of logic variables and of non-determinism.

Elimination of Logic Variables

Logic variables differ from variables in procedural languages by two main characteristics. The first is that they may be assigned only once; before the assignment they are said to be *free*, and afterwards they are said *bound* and may then only be read from. The second is that

when they are used as parameter to a predicate, it is generally not possible to determine by looking at the source program whether they will read or written by this predicate.

We have decided not to support logic variables in the generated prototypes because it is more efficient and more readable to replace them by normal variables instead of simulating their behaviour. This has two consequences: First, the single-assignment rule will not be enforced by the target language compiler, which may eventually lead to erroneous executions. Second, the profiles of CO-OPN methods will have to be precised by mode declarations, an operation which then becomes a part of the normal refinement process of the specifications. This is necessary because there exists no satisfying *mode inference* algorithm capable of determining unambiguously the modes of parameters simply by static analysis of the source text [Somogyi, Henderson&Conway 96]. The nature of our mode annotations is described in section 2.11.2.

Supporting Search Non-Determinism

The general way of supporting search non-determinism is by backtracking, a mechanism which is uneasy to reproduce in procedural languages. The problem is that it is not possible to implement directly the stack needed for remembering the choice points on the system stack of procedural languages [Aït-Kaci 91]: Whereas the stack of a procedural language shrinks when exiting from a function call, the stack of a logic language must keep the information relative to successful evaluations of non-deterministic predicates.

There are mainly two solutions for supporting the non-determinsitic features of CO-OPN:

- Either to make explicit the search procedure at the application layer. This means that the generated code will be full of loops and manipulation of iterators. This strategy may lead to very good performance, but is contrary to our general principles of software engineering mentioned above.
- Make non-deterministic primitives available to the user and hide the resolution mechanism. This induces a certain cost at run-time, but gives raise to much cleaner code at the application-level.

For the following axiom, belonging to an Object Divider and where `x` and `y` are variables of type Natural:

```
(y=0)=false => Divide : Dividend(x), Divisor(y) -> Division(x/y);
```

this is how we want the code to look like in Ada95:

```
1  procedure Divide (O: access Abstract_Divider; Result: in out Status) is
2      x : Natural := Natural_Prototype;
3      y : Natural := Natural_Prototype;
4  begin
5      Get(O.Dividend, x);
6      Get(O.Divisor, y);
7      if (y=0)/=false then return; end if;
8      Put(O.Division, x/y);
9      Result := Success;
10 end Divide;
```

Figure 109. Code Generated for Transition Divide

Let us comment this code. First, the transition is implemented as a procedure which returns a parameter `Result`, the role of which is to tell whether the execution failed or succeeded. By default it fails: This makes the code more succinct, as e.g. in line 7. The parameter `O` is a reference to the Object instance the transition belongs to: This gives us access to the places, which are private variables of the class `Abstract_Divider` defining our Object. Lines 2 and 3 declare the local variables. In Ada95, polymorphic variables must be initialized as soon as they are declared, hence the assignment to the value `Natural_Prototype`. Lines 5 and 6 retrieve the needed tokens from the input places of the transition. The `Get` procedure is non-deterministic, which means that if it fails then the caller must backtrack. Backtrack at the application-level is performed by exiting from the procedure. The underlying resolution mechanism will then prepare the environment in such a manner that when the procedure is called again, then the previous choice point will deliver its next solution. This is supported transparently in our implementation. Line 7 performs the test of the global condition. If it fails, then we must backtrack, and this is indicated to the resolution procedure by simply returning with `Status` assigned to its default value `Failure`. Line 8 executes the postcondition. The `Put` procedure is deterministic: It does not define a hidden choice point. Finally, at line 9, the execution is marked as successful.

There are two other kinds of non-deterministic operations to support: The random choice of one axiom among several, and the synchronization with other methods. For the first, we simply furnish a non-deterministic procedure called `ChooseAxiom` similar to the `Get`. For the second, we make available an asynchronous `Send` and a synchronous `Call` primitive.

Now, what does it take to support this transparent resolution mechanism? We need two things:

- The first is a way of transferring the control directly from a failed non-deterministic primitive into the resolution control layer, by short-circuiting the application-layer. This is done by resorting to exceptions, and although this may be a bit heavy, it achieves perfectly the desired transparency. Something similar could be achieved with the `setjmp` and `longjmp` functions available on most Unix platforms: Their effect is also to transfer the control transparently from a nested function call to a caller situated several levels down in the invocation hierarchy. Unfortunately, the `setjmp`/`longjmp` functions do not restore cor-

rectly the whole Ada environment: Exception handlers and other language-specific data are not taken into account by the `setjmp/longjmp` functions. On the other hand, the exception mechanism is predefined in Ada, and this means that all the information is preserved. We define an exception called `Backtrack` for this usage.

- The other mechanism is a hidden stack for storing the choice points and undoing side-effects upon backtracking. This stack is defined as an object-oriented class pattern: Each element of the stack is an object of the `Undoable` class or one of its heirs, i.e. an object which implements the procedure `Undo`. A similar construct can be found in [Gamma et al 95]. The `Undo` procedure takes no argument, it simply supposes that all the information needed for undoing an operation is already available inside the object. This allows the backtracking mechanism to undo any kind of operation stored on the stack, without having to know what the operation consists of. The `Undoable` objects are instantiated during the normal execution of all non-deterministic primitives, as well as those which produce side-effects, like the `Put` procedure seen above.

The backtracking mechanism works as follows:

1. If a procedure implementing a transition or a method exits due to the exception `Backtrack`, or terminates normally with its argument `Result` set to `Failure`, then the backtracking mechanism is started.
2. The last choice point, i.e. the top of the stack, or sometimes the one immediately below, is then marked as `To_Retry`.
3. A global boolean variable `Replay_Mode` is set to true.
4. A global pointer `Stack_Cursor` is reset to the bottom of the stack.
5. The application-level procedure is called again. Each primitive it calls which is non-deterministic or has side effects will inspect the variable `Replay_Mode` and find that it is set to true, meaning that a special behaviour is expected: Instead of initiating the search for a solution by pushing a new choice point on the stack, they must reuse the choice point they created during the normal execution before the backtrack. This choice point is given to them by the pointer `Stack_Cursor`. If this choice point is not marked by the flag `To_Retry`, they must simply *simulate normal execution* by returning the same result as the last time. This result was previously stored in the choice point. If on the other hand the flag `To_Retry` is set, then it means that the next solution must be searched for on the basis of the information in the choice point.
6. The pointer `Stack_Cursor` is advanced to the next choice point before the next primitive is called by the application layer.
7. When `Stack_Cursor` has reached the top of the stack, the variable `Replay_Mode` is reset to false, and normal forward execution starts again.

The point is of course to make as efficient as possible all operations executed when `Replay_Mode` is true, for instance by caching the previous results inside the choice point.

That way it is possible to reduce the cost of reexecuting a transition or method from the start at each backtrack.

6.5.2.3 Taking Concurrency into Account

Each transition and method of the specification is implemented as a procedure of the target programming language. In [Buchs&Hulaas 96] we presented another strategy where we used three smaller procedures, corresponding respectively to the precondition, synchronization, and postcondition of axioms(19). This had the disadvantage that it was difficult to share information, e.g. the variables of the axioms, between these three parts.

Execution at the application layer is seen as sequential. It is for instance not possible at that level to know if the Object is shared by several branches of a simultaneous synchronization, since all invocations are serialized. When the Object is in the position of emitting a simultaneity, then no tasks are used at the application layer to manage the parallel calls: It resorts to asynchronous message sending with `Send` and `Receive` primitives(20). For instance, the axiom `T WITH (M1(x)..M2(y)) & M3 : P1(x) -> P2(y)` where method `M1` takes an input parameter and `M2` an output parameter, is implemented as follows:

```
1  procedure T (O: access Abstract_Object; Result: in out Status) is
2      x : Natural := Natural_Prototype;
3      y : Natural := Natural_Prototype;
4      M1 : M1_Xid;
5      M2 : M2_Xid;
6      M3 : M3_Xid;
7      CurrEvent : MessageId;
8  begin
9      DeclareSync(Sim(Seq(M1,M2),M3));         -- Initialize synchronization
10     Get(O.P1, x);
11     Send(M1, x);                             -- Start thread of M1
12     Send(M3);                                -- Start thread of M3
13     loop
14         CurrEvent := NextEvent;              -- Wait for next event
15         if CurrEvent >= Reply(M1) then       -- Is it in the thread of M1 ?
16             if CurrEvent = Success(M1) then  -- Is it the reply from M1 ?
17                 Receive(M1);                 -- Get reply from M1
18                 Send(M2);                    -- Start M2
19             else                             -- It must be the reply from M2
20                 Receive(M2,y);               -- Get reply from M2
21             end if;
22         else                                 -- It must be the reply from M3
23             Receive(M3);                     -- Get reply from M3
24         end if;
25         if Done(M2) and Done(M3) then exit; end if;
26     end loop;
27     Put(O.P2, y);
28     Result := Success;
29 end T;
```

Figure 110. Code Generated for a Transition with Synchronization

19. The global conditions part is decomposed and distributed among the precondition and synchronization parts in order to check the conditions as soon as the needed variables are assigned.

20. A synchronous `call` primitive may also be furnished to cover the simple cases without simultaneity.

Let us comment this code. Lines 4 to 6 declare variables which serve as identifiers for the different branches of a synchronization. They are useful in many places, as shown below. The type `MessageId` of line 7 is predefined and constitutes the key in the control of incoming messages. Line 9 transmits to the control layers the structure of the future synchronizations. This allows preparing the synchronization vectors because the control layer cannot guess the nature of the emitted synchronizations simply by observing the ordering of `Send` and `Receive` primitives. The `DeclareSync` call also initializes the given identifiers. Lines 11 and 12 start the synchronization by sending the first requests of all branches of the simultaneity at the outermost level. The loop starting at line 13 is necessary since there is a simultaneity to manage. Line 14 waits for the next `Receive` event to deliver, and the role of the if-then-else structures of the following lines is to select the adequate action in correspondence with the received event. The operator "`>=`" returns true if the left-hand argument follows causally the right-hand argument because of a sequential synchronization operator expressed in the `DeclareSync` of line 8. Line 25 tests the condition for exiting from the loop.

There are several limitations relative to the usage of variables. At the specification level, it is not allowed for two threads of a simultaneity to assign the same variable, since the branches of a simultaneity are supposed to be independent and all variables to be assigned only once. It is not allowed either for one thread to assign a variable which is then read by another. This would induce a kind of stream-parallelism which we have chosen not to support. Other restrictions are listed in appendix C.1 on page 239.

6.6 Related Work

6.6.1 Fault-Tolerance in Executable Specifications and Logic Programming

Most approaches do throw-away prototyping, or limit themselves to sequential and centralized systems. There are therefore very few references to fault-tolerance in automatic implementations of formal specifications.

The RAPIDE formalism is used in [Kenney 96] for specifying and generating implementations for transaction processing in distributed database systems. In this context it seems obvious to reuse the two phase locking and commitment protocols, as Kenney does.

The European IPTES Project [IPTES 94] describes the incremental prototyping of distributed real-time systems, and even introduces the appellation *distributed prototyping* for this activity, but does not address the problem of fault-tolerance.

We know of no fault-tolerant implementations for the Prolog language, although this is a rather natural extension if we consider logic programming as belonging to the more general *relational* paradigm, like classical database systems. In Delta-Prolog [Pereira et al 86] the notion of transaction is used in order to atomically reconfigure distributed computation trees. This allows optimizing the backtracking mechanism by lowering as much as possible the priority of the faulty process, and thus limiting the negative effects of a conflict. The objective is however not fault-tolerance, but mere atomicity.

6.6.2 Distributing Strong Synchronous Systems

All current implementations of strong synchronous systems rely on a notion of clock which synchronizes the execution of sets of processes. Processes may be distributed along a single bus or even in more general local area networks under the condition that the real-time constraints are still guaranteed. An exception is [Caspi&Girault 95] who present a Unix/ Internet implementation which relaxes the time constraint, as we do. Their approach is to make the processes synchronize by message passing and to let them execute with a maximal time lag of one period.

CO-OPN is not designed for the description of reactive systems because of its non-deterministic semantics. That is why we can afford to consider instantaneity simply as a conceptual view instead of a real temporal constraint.

6.6.3 Compilation of Petri Nets and Algebraic Petri Nets

There are relatively few research groups working with algebraic Petri nets [Reisig 91], although their expressivity is very attractive for modelling complex systems. This is probably due to the quasi-impossibility of analyzing and statically proving properties such as liveness: The only general method for verifying the behaviour of an algebraic Petri net is by simulation [Buchs 89]. The approach presented in this thesis is to our knowledge the first attempt at compiling algebraic Petri nets.

Coloured Petri nets [Jensen 81] and *high-level Petri nets* [Jensen&Rozenberg 91] are close to algebraic Petri nets, in the sense that they also allow symbolic markings as preconditions, i.e. the terms of the preconditions may contain variables which are instantiated by pattern-matching with the actual candidate tokens. The approach taken in [Ilié&Rojas 93] for a subclass of coloured Petri nets called *well-formed nets* is to consider the choice of input tokens as a succession of selections and intersections of groups of tuples, a method inspired from database systems. This method is certainly more efficient than ours when it comes to managing places with many tokens and complex dependencies between the preconditions. The preconditions in well-formed nets may consist in combinations of four predefined operations which have the property of having easily computable inverses. Their algorithm does however not allow the orthogonal expression of the preconditions w.r.t. to the search mech-

anism, as we do, since their code is not intended to be read or modified. Another interesting contribution, applied to high-level nets, is [Bañares, Muro-Medrano&Villaroel 93] which uses techniques taken from artificial intelligence with the objective of maintaining up-to-date sets of tokens matching the given preconditions. The result is that the computation is not restarted from scratch each time the firing of a transition is attempted, since the previously found candidates are remembered between each firing. This approach is also more efficient than ours, but it is rather interpreted by nature, since the inference mechanism needs to manipulate the preconditions as first-class objects. Again, the purpose is not to provide implementations close to the imperative paradigm as we do.

For an overview of other approaches to implementation and simulation of Petri nets, we refer to the survey in [Taubner 87] and the more recent [Kordon 92]. The reference [Colom, Silva&Villaroel 86] appears to be the first serious attempt at implementing Petri nets in high-level programming languages.

In concurrent and distributed implementations of Petri nets, most approaches attempt to achieve better performance by automatically splitting the nets into smaller parts and adequately placing these sub-nets on the available processors [Taubner 87], [Chiola&Ferscha 93], [Bréant&Pradat-Peyre 94]. In the most extreme cases, each transition is executed by one dedicated task. In our sense this approach is more related to simulation than to real application implementation. Rather, as in [Kordon 92], we take the Petri net model as a behavioural and architectural specification of the software to implement. We think that it is necessary to keep the structure of the model in the generated code, if we want the developer to recognize the entities of the specification and to be able to exploit the benefits of our incremental prototyping process. Moreover, modular Petri net formalisms, such as CO-OPN, facilitate the description of complex systems as small, loosely coupled, modules, whereas classical Petri nets are unstructured.

[Gustavson&Törn 94] and [Holvoet&Verbaeten 95] exploit the object paradigm in order to generate high-level implementations: They define for instance both transitions and places as classes of the target language. Their objective is however not prototyping at the level of the generated code.

6.7 Epilogue

Contributions

Let us establish a summary of the contributions of this chapter:

- We have defined a framework for the incremental prototyping of automatically generated

distributed prototypes based on formal specifications. Mixed prototyping in its original form was never applied to other formalisms than algebraic abstract data types. This constitutes therefore a completely new experience, and that is why there is no related work on this topic.
- We have introduced a notion of support for transparent fault-tolerance in the field of automatic implementations of formal specifications and in logic (Prolog) programming.
- We have demonstrated a way of generalizing distributed implementations of strong synchronous systems by relaxing the time constraint and concentrating only on transmitting and simulating the notion of simultaneity.
- We have shown a way of compiling algebraic Petri nets, as CO-OPN is a super-set of this formalism.

Finally, designing a distributed implementation for CO-OPN constitutes in itself an achievement !

Conclusion and Future Work

The purpose of this chapter was to show that our prototyping technique may be reasonably applied to specification languages which are designed for expressing non-deterministic, dynamic and concurrent behaviours. The particular case of CO-OPN Objects was chosen, and this has several implications: CO-OPN is a very rich formalism, and it was therefore necessary to provide answers for combining all the three kinds of above-mentioned behaviours. This has been achieved with only minor restrictions to the source language.

An important question to clarify is whether OOMP really gives the opportunity for improving significantly the performance of the generated prototypes. There are two levels to take into account:

- The local performance of each Object may be greatly ameliorated by removing non-determinism and by prototyping the algebraic data types part.
- The performance then depends essentially on the quantity of distributed backtracking needed at run-time, which in turn depends on the nature of the application, i.e. whether it is search oriented and whether there are many shared Objects, which necessarily entails conflicts. Unfortunately this is the part of the prototypes that we have chosen, at least temporarily, not to let the developer infringe on.

Much work has been done in order to develop an economic stabilization procedure. More could be done in order to ameliorate the distributed resolution layer. We have chosen to minimize compile-time analysis of the specifications, since our primary goal was to create the most general control algorithms. Now that this has been achieved, it might be interesting to analyze statically different intra-Object and inter-Object dependencies in order to optimize our basic algorithms. It must also be noted that CO-OPN is a research language and, as

thus, subject to evolution. This is already perceptible in CO-OPN/2, where the concept of Object reference may greatly alleviate the distributed stabilization procedure.

Chapter 7

Prototyping the Object Application Layer

7.1 Introduction

In this chapter we will concentrate on the possibilities given to the programmer for completing or optimizing prototypes generated from CO-OPN Object specifications. General problems related to the choice of the target programming language or to the implementation of generic specification modules have already been treated in chapter 5, and will not be recalled here.

7.2 General Mapping Rules

The Objects of the source specifications are mapped as follows into the programming language Ada95:

- The Object construct of CO-OPN constitutes in itself a module and is therefore implemented in a separate compilation unit for the definition of the abstract class and has one child unit for each symbolic and concrete class.
- Mapping (compared to the general class pattern of figure 6 on page 19):
    1. There are two `constructors`, called `Create` and `Copy`, which are not visible at the specification level since Objects may not be instantiated dynamically in our version of CO-OPN.
    2. There are no `accessors` since Objects are encapsulated entities in CO-OPN. The names of the places are not publicly visible: The associated `Put` and `Get` primitives might be considered as private accessors since places are viewed as abstract data types by the Object itself.

3. The `operations` are the specified methods and transitions and each one is implemented by a procedure playing the role of a predicate. There is always an automatically generated function called `Initialize` which sets the Object to the specified initial marking.

7.3 Range of Action in the Prototyping of Objects

We give below some possibilities for prototyping CO-OPN Objects, and review them in more detail in the following sub-sections. The developer may for instance:

- Modify the data structures, i.e. change the internal representation of places.
- Optimize the search for solutions by eliminating non-determinism or detecting earlier the conditions which will lead an event to fail.
- Change the sequence in which synchronizations are emitted in order to increase or to reduce concurrency.
- Implement some optimizations which are (momentarily) not performed by the specification compiler, such as the recognition of common subexpressions.

7.3.1 Changing the Implementation of Places

Places are by default implemented as unbounded multi-sets. This is the most general solution, but this flexibility is costly and rarely needed. The developer may in many situations know that a given place will never contain more than a certain number of tokens, and thus use a fixed size array instead.

We would therefore like to allow the developer to change the internal representation of places. He may perform this by providing a new concrete sub-class derived from a predefined abstract class called `Abstract_Place`. This class hierarchy defines data structures which may contain any type of token. In order to prevent type errors at the application layer, e.g. by inserting a token of type *natural* in a place supposed to hold only *boolean* values, there exists a predefined generic package called `Typed_Places_G`. This package defines (by a hidden type composition) a strongly typed interface for the places actually implemented by sub-classes of `Abstract_Place`. It is thus only allowed to `Put` and `Get` correctly typed tokens.

When defining new implementations for places, it is important to take into account the kind of concurrency allowed by the control layers. If intra-object concurrency is permitted then it is necessary to protect the places and tokens by programming exclusive access mechanisms. There is currently no intra-object concurrency in CO-OPN prototypes since simultaneous

requests are serialized and stabilizations are sequential. The places are therefore easier to implement.

7.3.2 Guiding the Search

Two interesting opportunities for optimizing the search which is inherent to the execution of CO-OPN transitions and methods are:

- To schedule differently the various tests and `Get` predicates in order to limit the search space and to determine as early as possible whether the search will fail or not. Some criteria, that the specification compiler would be unable to apply, are to work in priority on places which are known to contain few tokens or to delay the evaluation of conditions which are weakly constraining or expensive to compute. This kind of optimization is known as *variable-value ordering heuristics* in the field of constraint satisfaction [Dechter&Meiri 89].
- To use more deterministic predicates in order to limit the amount of backtracking and choice point instantiations. We have experimented with `Get` predicates which iterate on the contents of places and perform themselves the needed tests on the token values so that the result is known to already satisfy most of the caller's conditions. In order to favour encapsulation and code reuse, these `Get` predicates should be defined as generic iterators which are called or instantiated with as argument the function which implements the pattern-matching and testing operations. In the same spirit, we show below that is possible to remove the non-deterministic choice of axiom to execute and to replace it by procedural control structures. These optimizations may probably partly be implemented by the specification compiler itself.

7.3.3 Modifying the Synchronizations

It is possible to change the order in which synchronizations are emitted in order to increase or to reduce concurrency as long as the semantics of the specification is respected. For instance, alternatives are by default executed serially until a successful reply is received. The developer could decide to parallelize the sending of alternatives and abort all remaining branches once the first successful reply is received. He may also change the ordering of the alternatives in order to try first the ones he suspects of having the highest probability of success.

7.4 An Example of Prototyping

7.4.1 The Abstract Class for Object DAL

We give here the Ada95 interface of the automatically generated abstract class for Object DAL. The interesting parts of the abstract class body will be explained in the next subsection. The complete specification of Object DAL may be found in appendix D on page 257.

```
with Root_Object_Pkg; use Root_Object_Pkg;
with Typed_Places_G;

with ADT_ListEvent; use ADT_ListEvent;
with ADT_Event; use ADT_Event;
with ADT_Action; use ADT_Action;

package Object_DAL is

   type Abstract_DAL is abstract new Root_Object with private;
   subtype DAL is Abstract_DAL'Class;
   type DAL_ref is access all DAL;

   --/ Public class methods:
   ------------------------
   --/ Initialization routine for the selection of a concrete implementation:
   procedure Set_DAL_Prototype is abstract;
   --/ Show the prototype object:
   function DAL_Prototype return DAL;
```

Figure 111. Public Type Declarations and Class Methods for DAL

This first part is quite similar to the abstract class of an Adt module. The predefined `Root_Object` is an abstract class which ensures that all Objects belong to the same root type. The following is different since there is no notion of Object instantiation at the specification-level in our version of CO-OPN.

```
   --/ Functions for constructing, copying and initializing:
   ---------------------------------------------------------
   --/ Virtual constructor and low-level initialization:
   function Create (Prototype: in Abstract_DAL) return DAL_ref is abstract;
   function Create return DAL_ref;

   --/ Copying function:
   function Copy (O: in Abstract_DAL) return DAL_ref is abstract;

   --/ High-level initialization (e.g. initial marking):
   procedure Initialize (O: access Abstract_DAL);
```

Figure 112. Functions for Constructing, Copying and Initializing DAL

The `Create` and `Copy` functions are abstract here since they depend on the internal representations of the concrete classes. The `Create` function is to instantiate the subcomponents of the Object: The internal representation of the different places is chosen here by initializing

them with the adequate prototype objects. The above declared `Initialize` procedure receives an empty body in the case of Object DAL because there is no specified initial marking.

The following is the list of specified methods of DAL. These are the main parts upon which incremental prototyping is applied. Notice that the profiles are identical to the specification, with the addition of parameter `O` on the left and parameter `Result` on the right.

```
--/ Specified methods and transitions:
procedure Act (O: access Abstract_DAL; a: in Action; Result: in out status);
procedure DisplayModif (O: access Abstract_DAL; l: in ListEvent; Result: in out status);
procedure DisplayConflict (O: access Abstract_DAL; a: in Action; Result: in out status);
procedure Transmit (O: access Abstract_DAL; a: in Action; Result: in out status);
procedure Confirm (O: access Abstract_DAL; a: in Action; Result: in out status);
procedure Conflict (O: access Abstract_DAL; a: in Action; Result: in out status);
```

Figure 113. Specified Methods of DAL

The private definitions concern the management of the prototype object and the instantiation of strongly typed places (to prevent the developer from inadvertently inserting wrongly typed tokens in a place). The complete definition of `Abstract_DAL` contains the places of the Object.

```
private
   --/ Initialize the class hierarchy:
   procedure Set_Hierarchy_Prototype (With_Prototype: in DAL_ref);

   --/ Definitions for strongly typed places:
   package ListEvent_Places is new Typed_Places_G(ListEvent); use ListEvent_Places;
   subtype ListEvent_Place is ListEvent_Places.Typed_Place;
   package Action_Places is new Typed_Places_G(Action); use Action_Places;
   subtype Action_Place is Action_Places.Typed_Place;

   --/ Complete view of the type:
   type Abstract_DAL is abstract new Root_Object with record
      confirmed: ListEvent_Place;
      conflicts: Action_Place;
      wait_transmit: Action_Place;
   end record;

end Object_DAL;
```

Figure 114. Private Definitions of Abstract_DAL

7.4.2 Prototyping of Method Act in Object DAL

Method `Act` is specified as follows (appendix D on page 257 gives a complete context):

```
ActOk :: Consistent(a,l)=true => Act a WITH Consult(l) : -> wait-transmit a;
ActNotOk :: Consistent(a,l)=false => Act a WITH Consult(l) : -> conflicts a;
```

The developer provides the following mode declarations for methods `Act` and `Consult`:

```
Act _ : action -> IN;
Consult _ : listevent -> OUT;
```

This is the automatically generated implementation of `Act`. Notice that it is totally non-deterministic (all predefined predicates are listed in appendix C.2 on page 244):

```
with Axiom_Enumeration_G;

with ADR_Client; use ADR_Client;
package body Object_DAL is

   type Act_Axiom_Name is (ActOk,ActNotOk);
   package Act_Axiom_Enumeration is new Axiom_Enumeration_G(Act_Axiom_Name);
   use Act_Axiom_Enumeration;

   procedure Act (O: access Abstract_DAL; a: in Action; Result: in out status) is
     l: ListEvent := ListEvent_Prototype;                -- Local variable from specification
     Consult: Consult_Xid;                               -- Transaction identifier for method call
     ChosenAxiom: Act_Axiom_Name;                        -- For non-deterministic choice of axiom
   begin
     DeclareSync(Consult);                               -- Tell the kind of synchronization needed
     ChooseAxiom((ActOk,ActNotOk),ChosenAxiom);          -- Arbitrary choice of axiom
     case ChosenAxiom is
       when ActOk =>
          call(ADR,Consult,l);                           -- Call method Consult of object ADR
          if Consistent(a,l)/=true then return; end if;  -- Verify global condition of axiom ActOk
          put(O.wait_transmit,a);                        -- Produce token of postcondition
       when ActNotOk =>
          call(ADR,Consult,l);                           -- Call method Consult of object ADR
          if Consistent(a,l)/=false then return; end if; -- Verify global condition of axiom ActNotOk
          put(O.conflicts,a);                            -- Produce token of postcondition
     end case;
                                                         -- At last we know the call is successful (by
     Result := success;                                  -- default a method call fails)
   end Act;
```

Figure 115. Automatically Generated Code for Method Act

The following hand-written version eliminates the non-deterministic choice of axiom to execute by exploiting the fact that the global condition discriminates between the two respective situations:

```
   procedure Act (O: access Concrete_DAL; a: in Action; Result: in out status) is
     l: ListEvent := ListEvent_Prototype;                -- Local variable from specification
     Consult: Consult_Xid;                               -- Transaction identifier for method call
   begin
     DeclareSync(Consult);                               -- Tell the kind of synchronization needed
     call(ADR,Consult,l);                                -- Call method Consult of object ADR
     if Consistent(a,l)=true then                        -- The global condition determines the axiom
         put(O.wait_transmit,a);                         -- Produce token of ActOk postcondition
     else
         put(O.conflicts,a);                             -- Produce token of ActNotOk postcondition
     end if;
                                                            -- At last we know the call is successful
     Result := success;                                  -- (by default it is considered as failed)
   end Act;
```

Figure 116. Concrete Implementation of Method Act

7.4.3 Possible Extensions

The weakness of the current scheme is that the developer may know a lot about the local dependencies which determine the flow of control, and he has no way of telling the resolution layer about them. We would therefore like to give a means for him to optionally transmit such information without cluttering too much the code. Most importantly, the optimizations must be compatible with the lower-level control mechanism. This means in particular that there must be no backward branching in the code: That would short-circuit the predefined backtracking mechanism, because the user-level execution would be completely desynchronized with respect to the internal stack which contains all the information about the current state of the search.

Let us take the following axiom, where places `P1` and `P3` both never contain more than 1 token:

```
(x=y)=false && (x=z)=false && (y=z)=false => t: P1(x),P2(y),P3(z) ->;
```

This is the corresponding translation produced by the prototyping tool:

```
get(O.P1,x);
get(O.P2,y);
if (x=y)/=false then return; end if;                    -- Backtrack to choice of y
get(O.P3,z);
if (x=z)/=false then return; end if;                    -- Backtrack to choice of z
if (y=z)/=false then return; end if;                    -- Backtrack to choice of z
```

If the condition `(x=z)/=false` is true then we should try to inform that the whole transition is deemed to fail, and if `(y=z)/=false` is true then we should select another value for `y` and not for `z`. We see essentially two ways of achieving this: introducing either the notion of nogoods or the possibility of labelling the choice points.

Optimizing with Nogoods

Let us try to introduce the notion of *nogood* at the user level. Nogoods have been used in various fields of artificial intelligence [Stallman&Sussman 77] [Dechter 90] and are combinations of values which must not be reproduced during the search because they violate some given constraints. The dependency relations between the different values may however be very complicated and not allow a simple way of expressing them. For instance, it is necessary to use the place names to tell when the current combination of tokens is bad, because a token value alone does not determine which place it came from.

```
get(O.P1,x);
get(O.P2,y);
if (x=y)/=false then nogood(O.P1,O.P2); return; end if;     -- Useful iff we backtrack to x
get(O.P3,z);
if (x=z)/=false then nogood(O.P1,O.P3); return; end if;     -- Backtrack to choice of x
if (y=z)/=false then nogood(O.P2,O.P3); return; end if;     -- Backtrack to choice of y
```

This works quite well, as long as no more than one token is needed from the same place. Another advantage is that they may be reused in forward mode: If the `get(O.P2,y)` eventually fails, then execution will retry the `get(O.P1,x)`. Let us take the variant where it succeeds because there are several identical tokens left in `P1`. When the `get(O.P2,y)` is reexecuted in forward mode, the search mechanism will exploit the previous nogoods and choose a token in `P2` which respects all the constraints recorded. In other words, the same errors are not repeated. This is referred to as *learning while searching* [Dechter 90] in the artificial intelligence community. The nogoods may however necessitate tremendous amounts of memory, since they represent all the intermediate failures in a search. Although some techniques exist to bound the size of the nogood sets, we will rather consider the next proposition, because it is more general.

Labelling Choice Points

Whereas the nogoods help selecting the choice point to return to, we could directly ask the programmer to label each choice point in order to work with immediate information. This would necessitate a new set of `get` primitives having an optional parameter which communicates the label of the current choice point to the resolution layer. In case of failure it would then be possible tell exactly where to backtrack to. The label could consist of a set of natural numbers or a locally defined enumerated type, which, in very strongly typed languages like Ada, would require a conversion to the type `natural`. Two predefined labels are needed: One for each of the cases where the target choice point is respectively unknown (by default control returns to the previous choice point) and when it is the caller (when the failure is known to be definitive).

```
procedure t (O: access Concrete_O; Return_to: in out Choice_Point; Result: in out Status) is
   type Transition_t_CP is (Choose_x,Choose_y,Choose_z);
   function CP is new Unchecked_Conversion(Transition_t_CP,Choice_Point);
   x,y,z : Natural := Natural_Prototype;
begin
   get(Choose_x,O.P1,x);
   get(Choose_y,O.P2,y);
   if (x=y)/=false then return; end if;                                  -- Backtrack to Choose_y
   get(Choose_z,O.P3,z);
   if (x=z)/=false then Return_to := Caller; return; end if;             -- Definitive failure
   if (y=z)/=false then Return_to := CP(Choose_y); return; end if;       -- Backtrack to Choose_y
   Result := Success;
end t;
```

Figure 117. Example with Labelling of Choice Points

This last proposition greatly ameliorates the run-time performance at a negligible cost while still remaining facultative. If we wanted to push the principle even further, it would be necessary to undo some of the mechanisms which were originally designed to hide the search, for instance the exceptions which are raised when there are no more untried tokens

in a place. Let us suppose, still in the same example, that we wanted to detect when place P2 is empty (in order to announce a definitive failure instead of backtracking to the choice of x):

```
get(Choose_x,O.P1,x);
get(Choose_y,O.P2,y,not_found);
if not_found then Return_to := Caller; return; end if;       -- Definitive failure
if (x=y)/=false then return; end if;                         -- Backtrack to last point: Choose_y
```

Until now we have only presented the problems relative to the preconditions. Let us briefly have a look at the possibilities for ameliorating the synchronization part. Here it should be possible to introduce some knowledge about the deterministic nature of the methods being called. The problem is that the programmer has only a local view of the situation as explained in 6.4.3.2. The underlying backtracking mechanism might be disturbed by the additional indications of the programmer and thus the completeness of the search would not be guaranteed anymore. Therefore it would require taking into account the static Object dependencies in order to determine the configurations where the programmer's indications may be safely followed. We have chosen not to support this kind of optimizations for the moment.

7.5 Automatic Verification of Concrete Classes

We showed in section 5.7 how to generate sub-classes of a given implementation of an algebraic abstract data type. This is useful for transparently testing at run-time whether the hand-written code conforms to the semantics of the specification. We give here some ideas about how a similar mechanism could be obtained in CO-OPN Objects. It should be noted that we do not address the problem of detecting global properties in distributed systems (we refer to [Garg 96] for a recent survey of that domain) since our objective is only to establish the correctness of the computation on a per Object basis. It is out of the scope of our executable assertion scheme to find errors in the coordination between concurrent Objects: Such problems are rather due to mistakes present in the specifications and should therefore be noticeable already when executing the abstract code. Let us also emphasize that even if the tests do not find any errors, it does not mean that the code is proved to be correct. There may still be errors which are not revealed by the particular execution of the prototype.

Even if we restrict ourselves to the information which is local to each Object, there are two major problems to answer:

- First, the search non-determinism implies that it is not possible to compare behaviours on the basis of a single execution of the abstract code: If the concrete code fires successfully a given event, the abstract code may well require several tentatives to deliver the same answer. The challenge is to perform in a cost-effective manner the simulation of the con-

crete code by the abstract code(1). It seems that the only general method is to abort the subtransaction enclosing the execution of the concrete code (but the results must be remembered) and then to execute repeatedly the abstract code until the same result is obtained. In [Hulaas 96] we proposed to represent methods and transitions by three procedures implementing respectively the precondition, synchronization and postcondition: This could allow the separate verification of sub-parts of the execution, a technique which would be less expensive.

- The second major problem is caused by the encapsulation: How can we compare the resulting Object states? Implementations of CO-OPN Objects do not logically provide any equality operator, because the source language does not need such comparisons. In our prototypes, we can fortunately circumvent this by directly comparing the sets of consumed and produced tokens, information which is available in the execution log, as explained in 6.5.2.2. This represents an important speed-up in the verification process.

This subject requires further research in order to establish partial verification techniques which are less expensive but at the same time provide significant comparison criteria. For instance it should be possible to check that only the specified preconditions are satisfied in the concrete code.

7.6 Epilogue

In this chapter we demonstrated some of the possibilities offered for prototyping CO-OPN Object implementations while still remaining faithful to the specifications. This part consitutes one of the most interesting contributions of this thesis, particularly the possibility of optimizing the search by removing non-deterministic behaviours in favour of more intelligent predicates or procedural control structures. Further work is to make the specification compiler generate such optimized code patterns while still ensuring legibility and safety.

We have also noticed that it is very difficult to test non-deterministic code without engaging into very expensive computations. Future research will concentrate on reducing this cost. The efficiency of the verification techniques are also very dependent on the openness of the generated code.

1. To test the bisimilarity (see section 3.7) it is necessary to show additionally that the concrete code can simulate the abstract code. Only this would guarantee that both versions result in the same set of target states.

Chapter 8
Conclusion

The original contributions of this thesis are placed on two levels. On the one hand, we propose a new methodology for the development of formally specified software, in particular for distributed systems. On the other hand, we bring some technical results relative to the algorithmics needed for putting distributed prototypes to work, and this ranges from compilation techniques for modular algebraic Petri nets to cooperative symbolic problem solving as well as automatic generation of programs with some support for fault-tolerance.

As a consequence, we may characterize our work as being rather *exploratory*, considering the limited amount of time that could be devoted to each specific subject. Software engineering is by nature a multi-disciplinary activity, which may explain that the contributions of this thesis result from a research that was conducted more “in breadth” than “in depth”. We think however that science must not only advance by specialization and deepening of individual domains, but also from time to time progress by cross-fertilization of different branches of research. We hope that our work will be considered as belonging to the latter kind of contribution.

8.1 Overview of Results

The objective of this thesis was to contribute to reducing the development cost of computer programs. To this end, we proposed a new prototyping methodology, *mixed prototyping with object-orientation*, the claimed advantage of which is to provide a smooth and safe transition from an implementation-near formal specification to a complete, efficient and maintainable realization in a main-stream object-oriented programming language.

Our approach is inspired from an existing methodology called *mixed prototyping*, from which it inherits all the benefits, both in terms of programmatic advantages, such as efficiency and correctness of the resulting software, and in terms of methodological progress,

such as higher flexibility in the development process and increased independence of software components, properties which enable the practice of concurrent engineering.

Our proposal is to adapt mixed prototyping to the object paradigm, of which we exploit the flexibility and inherent capability of modeling abstract entities, in order to make the prototyping process more intuitive and to increase the guarantee of correctness of the resulting implementation. This is how we justify our argument: The incremental prototyping process starts from a description of the software with the well-defined semantics of a formal specification language. The specification is then automatically translated into an initial implementation by the means of a certified correct specification compiler. The generated code may be ameliorated and completed manually by smaller increments than was possible before. During this phase, the abstract code is considered as a read-only: It is never modified and may in our approach be continuously taken as a reference implementation. This means that code reuse is promoted, and as well that the abstract code may be exploited for verifying that the hand-written code does not violate the semantics of the specification.

At the end of chapter 2 we established a set of objectives to be fulfilled by our prototyping methodology when applied to specific formalisms such as CO-OPN:

- It should be *intuitive*: This objective is perfectly attained, both in the algebraic data type part and the modular Petri net part of CO-OPN. The implementation remains faithful to the spirit of the formalism by mapping operations, transitions and methods into functions and procedures of the target programming language, and places and data structures into type definitions and variables. Moreover, the modular structure of the specification is preserved, and it is even possible to a certain extent to map specified generic modules into generic modules of the target language.
- It must lead to *correct implementations*: The compilation is based for the algebraic specifications on an existing algorithm which is proved correct. This is partly reused in the Object part of CO-OPN. Concerning the dynamic behaviour of CO-OPN, we do not provide a formal proof, but we strived to use as much as possible simple and intuitive algorithms, for which it is easier to convince oneself that they are correct.
- The generated code should be *efficient*. For AADTs, this is certainly true, although many additional optimizations can be exploited, as we show in chapter 4. For CO-OPN Objects, it is harder to emit any opinion, first because we lack elements of comparison, and, second, because we have not had the time to experiment with distributed prototypes. Anyway, it is in this part that the penalty for supporting the prototyping scheme is the highest. If we wanted to compare the performance with any other system, it would have to be with distributed process-based Prolog dialects, and even though they do not scale up with the complexity of CO-OPN, they do not have the reputation of being particularly efficient.
- The run-time support should be *robust* since we are not doing throw-away prototyping. For the distributed part we may say that this objective has been answered for by our implementation of synchronizations as nested transactions. For the distributed resolution

mechanism, we have already mentioned that we tried to use algorithms that are simple and thus less vulnerable than more optimized ones. We must admit that we are a bit worried about the memory consumption of the prototypes, since copies of object states are accumulated during synchronizations as a consequence of non-determinism and may not be released before the stabilization of the top-level event.

Seen from a larger perspective, we think that our work can considerably improve the quality of software implementations, and may thus contribute to a more widespread application of formal methods in the development process.

8.2 Limitations of our Approach

The weakness of OOMP resides, in our eyes, in three factors:

- Its inherent dependence upon the chosen target language. The range of possibilities offered by the prototyping process is bound to:
    1. The expressive power of the programming language. We have mentioned the problem of inheritance anomaly which hinders code reuse in concurrent applications.
    2. The structuring facilities of the target language. Ada95 seems for the moment to be the best candidate from that point of view.
    3. The efficiency of the target language. For improving the execution of non-deterministic code, it might be worthwhile to choose multi-paradigm languages, i.e. languages which integrate the logic paradigm with object-orientation and functional or procedural styles.

- The quality of the design patterns used for implementing a given source formalism. This defines the extent to which the developer may apply the incremental prototyping principles. AADTs are perfect from that point of view, since they may be modified in their integrality. The case of CO-OPN Objects is not so easy, since they are tightly coupled with a heavy run-time layer, which we have considered as non-prototypable. This decision was taken essentially because of the complexity of the algorithms, but it is possible that we were too cautious here. Another important factor is the level of support put to work just for enabling the prototyping process. We designed the resolution layer to be orthogonal to the application layer in order to support prototyping at the application-level. This induces a certain cost, but we think that this cost may be practically neglected in programs which do not need backtracking.

- Interfacing with existing software: The usage and initialization of prototype objects designed for limiting the impact of change during the OOMP process may make the

resulting software a bit complicated to interface with. When a prototype is considered as achieved it is however possible to remove this mechanism by editing the source files. The collaboration of AADT implementations with external software should then be straightforward. Distributed CO-OPN Objects may be much more difficult to interface with because of the concurrency control and the protocol for the distributed resolution. There are many problems of heterogeneity which must be solved before this can work and which are not especially related to OOMP, as for instance the nature of the communication, i.e. if it is performed by RPC or lower-level message-passing.

8.3 Perspectives and Open Problems

We did not have the time to complete the implementation of the specification compiler, the main problem being the algorithm for compiling pattern-matching. We had therefore to "simulate the code generation": Some abstract and symbolic classes have been coded by hand in order to experiment with the OOMP concept.

The incremental prototyping process has thus been successfully applied to algebraic abstract data types and to centralized algebraic Petri nets, which means for instance that the local part of the resolution layer works perfectly. However we did not have the time to play with complete distributed prototypes: The concurrency control mechanism has been validated on a network of Unix workstations, but not the distributed resolution layer.

The first thing to do would therefore be to achieve the implementation of the compiler and the distributed run-time support in order to evaluate the real performance of the distributed prototypes. Since we have systematically chosen to optimize execution for programs with little search non-determinism, it would be interesting to see to which extent the prototypes can rivalize with entirely hand-coded applications.

Since we have decided to support a maximal subset of the semantics of CO-OPN, a useful exercice would be determine whether it could lead to important improvements to suppress some of the language's constructs or properties. The most obvious constraints would be to eliminate synchronizations from internal transitions in order to simplify the stabilization process, and to let the developer reduce the need for backtracking by providing determinism annotations. Finally, it would be necessary to evaluate the impact on the modelling power these changes would have.

Of course, another interesting challenge would be to apply OOMP to new specification formalisms.

Bibliographic References

[Ada 83]
Reference Manual for the Ada Programming Language. ANSI/MIL-Std-1815a, 1983

[Ada 95]
Ada Reference Manual, Language and Standard Libraries, ISO/IEC 8652:1995

[Aït-Kaci 91]
H. Aït-Kaci, *Warren's Abstract Machine, A Tutorial Reconstruction*, The MIT Press, 1991

[André et al 96]
C. André, F. Boulanger, M.-A. Péraldi, J.-P. Rigault, G. Vidal-Naquet, *Objets et Programmation Synchrone*, Congrès AFCET "Modélisation des Systèmes Réactifs", Brest, France, pp. 55-62, March 1996

[Andrews 79]
D.M. Andrews, *Using Executable Assertions for Testing and Fault Tolerance*, FTCS 9, Madison, pp 102-105, 1979

[Asur&Hufnagel 93]
S. Asur and S. Hufnagel, *Taxonomy of Rapid-Prototyping Methods and Tools*, 4th International Workshop on Rapid System Prototyping, Research Triangle Park Institute, North Carolina, USA, June 1993

[Babaoglu&Toueg 93]
Ö. Babaoglu, S. Toueg, *Non-Blocking Atomic Commitment*, In "Distributed Systems" (2nd edition), S. Mullender (Ed.), ACM Press, New York, Addison-Wesley, 1993

[Bañares, Muro-Medrano&Villaroel 93]
J.A. Bañares, P.R. Muro-Medrano, J.L. Villaroel, *Taking Advantages of Temporal Redundancy in High Level Petri Nets Implementation*, Proceedings of Application and Theory of Petri Nets '93, Chicago, LNCS 691, pp. 32-48, 1993

[Barbey, Buchs&Péraire 97]
S. Barbey, D. Buchs, C. Péraire, *A Theory of Specification-Based Testing for Object-Oriented Software*, EDCC2, Taormina, Italy, To appear in LNCS, 1997

[Bergstra&Klint 96]
J.A. Bergstra, P. Klint, *The ToolBus Coordination Architecture*, In Procs. Coordination Languages and Models, Cesena, Italy, P. Ciancarini & C. Hankin (Eds.), LNCS Vol 1061, Springer, Berlin, 1996

[Bernstein, Hadzilacos&Goodman 87]
A.J. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Distributed Database Systems,* Addison Wesley, 1987

[Berry&Gonthier 88]
G. Berry and G. Gonthier, *The Esterel Synchronous Programming Language: Design, Semantics, Implementation,* Technical Report 842, INRIA, May 1988

[Biberstein&Buchs 95]
O. Biberstein, D. Buchs, *Concurrency and Object-Orientation with Structured Algebraic Nets,* G. Bernot & M. Aiguier editors, Working papers of the International Workshop on Information Systems - Correctness and Reusability - IS-CORE'95, Research Report, Université d'Evry, France, September 1995

[Biberstein 97]
O. Biberstein, *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems,* PhD thesis, University of Geneva, July 1997.

[Bidoit 89]
M. Bidoit, *Pluss, un langage pour le développement de spécifications algébriques modulaires,* PhD thesis, Université Paris-Sud, France, 1989

[Birell&Nelson 84]
A. Birell, B. Nelson, *Implementing Remote Procedure Call,* ACM Transactions on Computer Systems, Feb 1984

[Birman, Schiper&Stephenson 91]
K. Birman, A. Schiper, P.Stephenson, *Lightweight Causal and Atomic Group Multicast,* ACM Transactions on Computer Systems, 9(3), pp. 272-314, 1991

[Boehm 88]
B. Boehm, *A Spiral Model of Software Development and Enhancement,* IEEE Computer, 21(5):61-72, 1988

[Boniol&Adelanto 93]
F. Boniol, M. Adelantado, *Programming Distributed Reactive Systems: a Strong and Weak Synchronous Coupling,* 7th International Workshop on Distributed Algorithms WDAG'93, LNCS 725, Sept. 1993, pp 294-308

[De Bosschere 94]
K. De Bosschere, *Process-based Parallel Logic Programming: A Survey of the Basic Issues,* 11th ICLP: Post-Conference Workshop W1 on Process-Based Parallel Logic Programming, Sta. Margherita, Italy, June 1994

[Bréant&Pradat-Peyre 94]
F. Bréant, J.F. Pradat-Peyre, *An Improved Massively Parallel Implementation of Coloured Petri Nets Specifications,* IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Ascona, Switzerland, 1994

[Breu 91]
R. Breu, *Algebraic Specification Techniques in Object-Oriented Programming Environments,* LNCS vol 562, Springer Verlag, 1991

[Broy et al 93]
M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, K. Stølen, *The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0*, Technical Report TUM-I9311-2, Technische Universität München. Institut für Informatik, May 1993

[Buchs 89]
D. Buchs, *Ateliers de génie logiciel et spécification de logiciel*, PhD thesis, No. 2361, University of Geneva, 1989

[Buchs&Guelfi 91]
D. Buchs, N. Guelfi, *CO-OPN: A Concurrent Object Oriented Petri Net Approach for System Specification*, 12th International Conference on theory and application of Petri Nets, Aarhus, pp. 432-454, 1991

[Buchs, Flumet&Racloz 93]
D. Buchs, J. Flumet, P. Racloz, *SANDS Structured Algebraic Net Development System*, Research report no 71, CUI 1993, also in 14th Int. Conf. Th. Petri Nets, Tool Presentation, Chicago, 1993

[Buchs et al 95]
D. Buchs, J. Hulaas, P. Racloz, M. Buffo, J. Flumet, E. Urland, *SANDS Structured Algebraic Net Development System for CO-OPN*, 16th International Conference on Application and Theory of Petri Nets, Torino, Italy, 1995, pp. 45-53. Extended version available as technical report: D. Buchs, O. Biberstein, M. Buffo, C. Buffard, J. Flumet, J. Hulaas, G. Di Marzo, P. Racloz, *$SANDS_{1.5}$/CO-$OPN_{1.5}$: An Overview of the Language and its Supporting Tools*, Technical Report #95/133, DI-EPFL, 1995

[Buchs&Hulaas 95]
D. Buchs, J. Hulaas, *Incremental Object-Oriented Implementation of Concurrent Systems Based on Prototyping of Formal Specifications*, SIPAR workshop, Biel, Switzerland, pp. 141-145, Oct. 1995

[Buchs 96]
D. Buchs, *Méthodes formelles pour le développement et la vérification de logiciels*, 9èmes journées internationales, le génie logiciel et ses applications, In Revue Génie Logiciel, 18-21 Nov. 1996

[Buchs et al 96]
D. Buchs, P. Racloz, M. Buffo, J. Flumet and E. Urland, *Deriving Parallel Programs Using SANDS Tools*, Transputer Communication Journal, vol. 3, no. 1, 1996, pp. 23-32

[Buchs&Hulaas 96]
D. Buchs, J. Hulaas, *Evolutive Prototyping of Heterogeneous Distributed Systems Using Hierarchical Algebraic Petri Nets*, Procs. IEEE International Conference on Systems, Man and Cybernetics SMC'96, Beijing, China, Oct. 14-17 1996, pp. 3021-3026. Also available as: European Esprit Long Term Research Project 20072 "Design for Validation" (DeVa) technical report #09, 1996

[Buck et al 94]
J. Buck, S. Ha, E.A. Lee, D.G. Messerschmitt, *PTOLEMY: a Framework for Simulating and Prototyping Heterogeneous Systems*, International Journal of Computer Simulation, April 1994

[Buffo&Buchs 96]
M. Buffo, D. Buchs, *Contextual Coordination between Objects*, X SBES Brazilian Symposium on Software Engineering, JC. Maldonado&PC. Masiero (Eds.), Brazil, pp. 341-356, Oct. 1996

[Caspi&Girault 95]
P. Caspi, A. Girault, *Execution of Distributed Reactive Systems*, In Proceedings Europar'95, Stockholm, Sweden, LNCS Vol. 966, pp. 15-26, August 1995

[CCITT 88]
CCITT, Z.100, *CCITT Specification and Description Language*, 1988

[Chandra&Toueg 96]
T. Chandra, S. Toueg, *Unreliable Failure Detectors for Reliable Distributed Systems*, Journal of the ACM, Vol. 34, No 1, March 1996. A preliminary version appeared in ACM International Symposium on Principles of Distributed Computing (PODC'92), Aug. 1992

[Chang, Despain&DeGroot 85]
J.-H. Chang, A.M. Despain, D. DeGroot, *AND-Paralellism of Logic Programs Based on Static Data Dependency Analysis*, pp. 218-225 in Digest of Papers of COMPCON Spring '85, 1985

[Cherki&Choppy 96]
S. Cherki, C. Choppy, *Une méthode de rétroingénierie utilisant les spécifications algébriques*, Actes des Journées du GDR Programmation, Orléans, November 1996

[Chiola et al 91]
G. Chiola, C. Dutheillet, G. Francheschinis, S. Haddad, *On Well-Formed Coloured Nets and their Symbolic Reachability Graph*, LNCS : High-Level Nets, Theory and Application. K. Jensen, G. Rozenberg (Eds.), Springer, 1991

[Chiola&Ferscha 93]
G. Chiola, A. Ferscha, *Distributed Simulation of Petri Nets*, IEEE Parallel and Distributed Technology, Systems and Applications, Vol. I, No. 3, pp. 33-50, IEEE Computer Society Press, August 1993

[Choppy&Johnen 85]
C. Choppy, C. Johnen, *PETRIREVE: Petri Net Transformations and Proofs with Rewriting Systems*, In Procs. Sixth European Workshop on Applications and Theory of Petri Nets, 1985.

[Choppy 87]
C. Choppy, *Formal Specifications, Prototyping and Integration Tests*, Proc. of the 1st European Software Engineering Conference, 1987

[Choppy 88]
C. Choppy, *ASSPEGIQUE user's manual*, Technical Report No 452, LRI, Université Paris XI Orsay, 1988

[Choppy&Kaplan 90]
C. Choppy, S.Kaplan, *Mixing Abstract and Concrete Modules: Specification, Development and Prototyping*, Proc. 12th Int. Conf. on Software Engineering, Nice, 1990

[Choppy et al 89]
C. Choppy, S. Kaplan, V. Meissonnier, *Mixing executable specifications and code evaluation: An approach for prototyping and integration testing*, Proc. International Congress on New Technologies for Software and Supercomputers Development, Caracas, November 1989

[Clark 84]
K. Clark, S. Gregory, *Parlog: Parallel Programming in Logic*, Technical Report DOC 84/4, Imperial College, London, April 1984

[Coleman et al 94]
D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes, *Object-Oriented Development: The Fusion Method*, Prentice-Hall, NJ, 1994

[Colmerauer 83]
A. Colmerauer, H. Kanoui, M. van Caneghem, *Prolog, Theoretical Basis and Current Developments, Technique et Science informatiques*, Vol.2, No 4, pp. 271-311, July-August 1983

[Colom, Silva&Villaroel 86]
J.M. Colom, M. Silva, J.L. Villaroel, *On Software Implementation of Petri nets and colored Petri nets using High-Level Concurrent Languages*, 7th Workshop on Application and Theory of Petri Nets, pp. 207-241, Oxford, 1986

[Comon 89]
H. Comon, *Inductive Proofs by Specifications Transformation.* In N. Dershowitz, editor, Procs of the 3rd Intl Conf on Rewriting Techniques and Applications, vol 355 of LNCS, pp 76-91, Springer Verlag, Berlin, 1989

[Coplien 92]
J.O. Coplien, *Advanced C++ programming styles and idioms*, Addison-Wesley, 1992

[Dechter&Meiri 89]
R. Dechter, I. Meiri, *Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems*, Proc. of IJCAI-89, pp. 271-277, Detroit, MI, 1989

[Dechter 90]
R. Dechter, *Enhancement Schemes for Constraint Processing: Backjumping, Learning and Cutset Decomposition*, Artificial Intelligence, 41(3):273-312, 1990

[DeGroot 84]
D. DeGroot, *Restricted AND-Parallelism*, In International Conference on Fifth Generation Computer Systems, pp. 471-478, Tokyo, November 1984

[Dershowitz&Jouannaud 90]
N. Dershowitz, J.-P. Jouannaud, *Rewriting systems*, Handbook of Theoretical Computer Science, Van Leuwen editor, North Holland, 1990

[Drakos 90]
N. Drakos, *Sequential and Parallel Execution of Logic Programs with Dependency Directed Backtracking*, PhD thesis, Univeristy of Leeds, July 1990

[Dijkstra 71]
E.W. Dijkstra, *Hierarchical Ordering of Sequential Processes*, Acta Informatica 1, 1971, pp. 115-138

[Dijkstra 75]
E.W. Dijkstra, *Guarded Commands, Nondeterminacy, and Formal Derivation of Programs*, Communications of the ACM 18(8), pp. 453-457, August 1975

[Dixon et al 89]
G.N. Dixon, G.D. Parrington, S.K. Shrivastava, S.M. Wheater, *The Treatment of Persistent Objects in Arjuna*, Proc. European Conf. on Object-Oriented Programming ECOOP'89, July 1989

[Dürr&Plat 95]
E.H. Dürr, N. Plat (editor), *VDM++ Language Reference Manual*, Afrodite (ESPRIT-III project number 6500) document id. AFRO/CG/ED/LRM/V11, Cap Volmac, August 1995

[Ehrig&Mahr 85]
H. Ehrig and B. Mahr , *Fundamentals of algebraic specification 1: equations and initial semantics*, EATCS Monographs, Springler Verlag, 1985

[Eliëns 92]
A. Eliëns, *DLP - A Language for Distributed Logic Programming*, Wiley, 1992

[Elmstrøm 93]
R. Elmstrøm, *IPTES Toolset*, FME'93: Industrial Strength Formal Methods, Springer, Odense, 1993

[Eswaran et al 76]
K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger, *The Notion of Consistency and Predicate Locks in a Database System*, Comm. of the ACM, Vol.19, No. 11, pp. 624-633, 1976

[Ferrenczi&Futo 92]
Sz. Ferrenczi and I. Futo, *CS-Prolog: a Communicating Sequential Prolog*, in P. Kacsuk and M. Wise editors, "Implementations of Distributed Prolog", pp 357-378, John Wiley & Sons, Chichester, 1992

[Ferscha 96]
A. Ferscha, *Parallel and Distributed Simulation of Discrete Event Systems*, In "Parallel & Distributed Computing Handbook", A.Y.H. Zomaya (Ed.), McGraw-Hill, 1996

[Flumet 95]
J. Flumet, *Un environnement de développement de spécifications pour systèmes concurrents*. PhD thesis No 2761, University of Geneva, 1995

[Fröhlich&Larsen 96]
B. Fröhlich, P.G. Larsen, *Combining VDM-SL Specifications with C++ Code*, Formal Methods Europe '96, Oxford, March 1996

[Frølund 92]
S. Frølund, *Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages*, ECOOP'92 Conference Proceedings, O. Lehrmann Madsen (Ed.), July 1992

[Fromentin&Raynal 94]
E. Fromentin, M. Raynal, *Local States in Distributed Computations: A Few Relations and Formulas*, Operating Systems Review, Vol. 28, No 3, pp 4-15, 1994

[Gabriel 89]
Gabriel, *Draft Report on Requirements for a Common Prototyping System*, ACM Sigplan Notices, 24(3):32-41, 1989

[Gamma et al 95]
E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Ma, 1995

[Garavel 89]
H. Garavel, *Compilation of LOTOS Abstract Data Types*, Proceedings of 2nd FORTE, Vancouver, Canada, 1989

[Garavel&Turlier 93]
H. Garavel, Ph. Turlier, *CAESAR.ADT: un compilateur pour les types abstraits algébriques du langage LOTOS*, Actes du Colloque Francophone pour l'ingénérie des protocoles, CFIP '93, Montréal, Canada, 1993

[Garg 96]
V.K. Garg, *Observation of Global Properties in Distributed Systems*, IEEE International Conference on Software and Knowledge Engineering, Lake Tahoe, Nevada, pp. 418-425, June 1996

[Gaschnig 79]
J. Gaschnig, *Performance Measurement and Analysis of Certain Search Algorithms*, Technical Report CMU-CS-79-124, Carnegie Mellon University, 1979

[Gaudel 85]
M.-C. Gaudel, *Towards structured algebraic specifications*, Proc. ESPRIT Technical Week, Bruxelles, Springer Verlag, 1985

[Gaudel et al 96]
M.C. Gaudel, B. Marre, F. Schlienger, G. Bernot, *Précis de génie logiciel*, Masson, Paris, 1996

[Goguen, Jouannaud&Meseguer 84]
J. Goguen, J.-P. Jouannaud, J. Meseguer, *Operational Semantics for Order-Sorted Algebra*, Research Report 84-R-101, CRIN, 1984

[Goguen&Meseguer 87]
J. Goguen, J. Meseguer, *Order-sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Problems*, in IEEE Symp. on Logic in Computer Science, Ithaca-NY, 1987

[Goldberg 84]
A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1984

[Gravell&Henderson 96]
A. Gravell, P. Henderson, *Executing Formal Specifications need not be Harmful*, Software Engineering Journal, Vol 11, number 2, March 1996

[Gray 78]
J. Gray, *Notes on Database Operating Systems*, In "Operating Systems: An Advanced Course", LNCS Vol 60, Springer Verlag, 1978

[Guelfi 94]
N. Guelfi, *Les réseaux algébriques hiérarchiques: un formalisme de spécifications structurées pour le développement de systèmes concurrents*, Thèse de doctorat, No d'ordre 3313, Université ParisXI Orsay, 1994

[Guerraoui et al 92]
R. Guerraoui, R. Capobianchi, A. Lanusse, P. Roux, *Nesting Actions through Asynchronous Message Passing: The ACS Protocol*, Proc. European Conference on Object-Oriented Programming, Springer Verlag, Utrecht, 1992

[Guerraoui 93]
R. Guerraoui, *Nested Transactions: Reviewing the Coherence Contract*, Journal of Computer and Information Science, No. 3, North Holland, 1993

[Guerraoui 95]
R. Guerraoui, *Les langages concurrents à objets*, Techniques et Sciences Informatiques, Vol. 14, No. 8, 1995

[Guerraoui&Schiper 95]
R. Guerraoui, A. Schiper, *The Decentralized Non-Blocking Atomic Commitment Protocol*, Proc. IEEE Intl. Symposium on Parallel and Distributed System Processing (SPDP-7), San Antonio, Texas, Oct. 1995

[Gustavson&Törn 94]
Å. Gustavson and A. Törn, *XSimNet, a Tool in C++ for Executing Simulation Nets*, In Procs. of European Simulation Multiconference ESM'94, Barcelona, Spain, June 1-3, pp. 146-150, 1994

[Guttag&Horning 78]
J.V. Guttag, J.J. Horning, *The Algebraic Specification of Abstract Data Types*, Acta Informatica, 10:27-52, 1978

[Guttag, Horning&Wing 85]
J. Guttag, J.J. Horning, J.M. Wing, *The Larch family of specification languages*, IEEE Software, vol. 2, no. 5, pp. 24-36, 1985

[Hallmann 91]
M. Hallmann, *A Process Model for Prototyping*, Software Engineering and its Applications, Toulouse, 9-13 December 1991

[Hansson et al 90]
H. Hansson, B. Jonsson, F. Orava, B. Pehrson, *Specification for Validation*, Formal Description Technique, II, Elsevier, North Holland, pp. 227-245, 1990

[Harder&Reuter 83]
T. Harder, A. Reuter, *Principles of Transaction-Oriented Database Recovery*, ACM Computing Surveys, Vol. 15, No. 4, 1983

[Harel&Pnueli 85]
D. Harel, A. Pnueli, *On the Development of Reactive Systems*, In "Logic and Models of Concurrent Systems, NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems", Springer Verlag, 1985

[Harel 87]
D. Harel, *Statecharts: A Visual Formalism for Complex Systems*, Science of Computer Programming, 8(3):231-275, 1987

[Harel 90]
D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot, *STATEMATE: A Working Environment for the Development of Complex Reactive Systems*, IEEE Transactions on Software Engineering, 16(4):403-414, April 1990

[Harper 86]
R. Harper, *Introduction to Standard ML*. Technical Report ECS-LFCS-86-14, University of Edinburgh, Dept. of Computer Science, November 1986

[Hartmanis 94]
J. Hartmanis, *Turing Award Lecture: On Computational Complexity and the Nature of Computer Science*, Communications of the ACM, Vol. 37, No 10, October 1994

[Henderson, Somogyi&Conway 96]
F. Henderson, Z. Somogyi, T. Conway, *Determinism Analyis in the Mercury Compiler*, In Proceedings Australian Computer Science Conference, Melbourne, Australia, pp 337-346, 1996

[Hennicker&Schmitz 96]
R. Hennicker, C. Schmitz, *Object-Oriented Implementation of Abstract Data Type Specifications*, 5th Intl. Conference on Algebraic Methodology and Software Technology AMAST'96, Wirsing&Nivat (Eds.), Munich, Germany, July 1996

[Hermenegildo 86]
M.V. Hermenegildo, *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*, PhD thesis, The University of Texas at Austin, 1986

[Hermenegildo&Rossi 93]
M.V. Hermenegildo, F. Rossi, *Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions*, Journal of Logic Programming, 12:1-199, pp 1-44, Elsevier, 1993

[Hoare 72]
C.A.R. Hoare, *Proof of correctness of data representations*, Acta Informatica 1, pp 271-281, 1972

[Hoare 78]
C.A.R. Hoare, Communicating Sequential Processes, Communications of the ACM, 21(8), pp. 666-677, August 1978

[Holvoet&Verbaeten 95]
T. Holvoet, P. Verbaeten, *PN-TOX: a Paradigm and Development Environment for Object Concurrency Specifications*. In Procs. Workshop on Object-Oriented Programming and Models of Concurrency, ICATPN'95, Torino, Italy, 1995

[Hsiang&Srivas 85]
J. Hsiang, M.K. Srivas, *A PROLOG Environment Developing and Reasoning about Data Types*, TAPSOFT, March 1985, pp 276-293, Berlin, 1985

[Hudak, Peyton Jones&Wadler 92]
P. Hudak, S. Peyton Jones, P. Wadler, eds. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)*, ACM SIGPLAN Notices, May 1992

[Hulaas 95]
J. Hulaas, *Introduction to Object-Oriented Mixed Prototyping with Algebraic Specifications*, Technical Report #95/155, DI-EPFL, October 1995.

[Hulaas 96]
J. Hulaas, *An Evolutive Distributed Petri Nets Simulator*, 10th European Simulation Multiconference ESM'96, Budapest, Hungary, 2-6 June 1996, pp. 348-352.

[Ichisugi, Matsuoka&Yonezawa 92]
Y. Ichisugi, S. Matsuoka, A. Yonezawa, *Rbc1: A Reflective Object-Oriented Concurrent Language without a Run-Time Kernel*, In Proc. Intl. Workshop on New Models for Software Architecture and Protocols'92; Reflection and Meta-Level Architecture, pp. 24-35, 1992

[Ilié&Rojas 93]
J.M. Ilié, O. Rojas, *On Well-Formed Nets and Optimizations in Enabling Tests*, in "Application and Tehory of Petri Nets '93", 14th International Conference, Chicago, LNCS Vol 691, M.A. Marsan (Ed.), June 1993

[INMOS 88]
INMOS, *OCCAM 2 Reference Manual*, Prentice-Hall, 1988

[IPTES 94]
The IPTES Consortium, *Overview of IPTES Results*, R. Elmstrøm (Ed.), Technical Report IPTES-IFAD-263-V2.0 of European ESPRIT "Incremental Prototyping Technology for Embedded Real-Time Systems" (IPTES) Project EP5570, January 1994

[ISO 84]
ISO IS 7498, Information Processing Systems, Open Systems Interconnection, *Basic Reference Model*, 1984

[ISO 88]
ISO IS 8807, Information Processing Systems, Open Systems Interconnection, *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, July 1988

[Jensen 81]
K. Jensen, *Coloured Petri Nets and the Invariant Method*, T.S.C. 14, pp. 317-336, 1981

[Jensen&Rozenberg 91]
K. Jensen and G. Rozenberg (Eds.), *High-Level Petri Nets*, Springer Verlag, 1991.

[Jones 90]
C.B. Jones, *Systematic Software Development Using VDM (second edition)*, Englewood Cliffs, NJ, Prentice-Hall, 1990

[Kaplan 87]
S. Kaplan, *A Compiler for Conditional Term Rewriting Systems*, In Proceedings of Rewriting Techniques and Applications '87, Bordeaux, LNCS Vol 156, Springer, Berlin, 1987

[Karsenty 96]
A. Karsenty, *GroupDesign: un collecticiel synchrone pour l'édition partagée de documents*, PhD Thesis, University of Paris XI Orsay, France, 1996

[Kemmerer 90]
R.A. Kemmerer, *Integrating Formal Methods into the Development Process*, IEEE Software, Vol. 7, no 5 (Sept):37-50, 1990

[Kenney 96]
J.J. Kenney, *Executable Formal Models of Distributed Transaction Systems Based on Event Processing*, PhD Thesis, Stanford University, June 1996

[Kiczales 96]
G. Kiczales, *Beyond the Black Box: Open Implementation*, IEEE Software, Vol. 13, no. 1 (Jan.): 8-11, 1996

[Kaplan 87]
S. Kaplan, *A Compiler for conditional term rewriting systems*, Proc. Rewriting Techniques and Applications 87, Bordeaux, Lecture Notes Computer Science 256, Springer, Berlin, 1987

[Kernighan&Ritchie 78]
B.W. Kerighan, D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978

[Knuth&Bendix 70]
D.E. Knuth, P. Bendix, *Simple Word Problem in Universal Algebras.* In J. Leech, editor, Computational Problems in Abstract Algebra, pp. 263-297, Pergamon Press, Oxford, 1970.

[Kordon 92]
F. Kordon, *Prototypage de systèmes parallèles à partir de réseaux de Petri colorés, application au langage Ada dans un environnement centralisé ou réparti*, PhD Thesis, Université Pierre et Marie Curie (Paris VI), Rapport No 92/34, Institut Blaise Pascal, Laboratoire MASI, 1992

[Kordon 94]
F. Kordon, *Formal Techniques Based on Nets, Object Orientation and Reusability for Rapid Prototyping of Complex Systems*, In Procs IFIP-WG 10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Ascona, Switzerland, April 1994

[Kordon 95]
F. Kordon, *H-COSTAM: a Hierarchical Communicating State-machine Model for Generic Prototyping*, Procs. 6th Intl. Workshop on Rapid System Prototyping, Triangle Park Institute, N. Kanopoulos (Ed.), IEEE Computer Society Press, pp 131-138, June 1995

[Krief 92]
Ph. Krief, *Utilisation des langages objets pour le prototypage*, Masson, Paris, 1992

[Kumar&Lin 86]
V. Kumar, Y.-J. Lin, *An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs*, In Procs. 3rd Int. Conf. on Logic Proramming, E. Shapiro (Ed.), Springer Verlag, London, pp. 55-68, 1986

[Lamport 78]
L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM, Vol. 21, No. 7, pp. 558-565, July 1978

[Lin 93]
H. Lin, *Procedural Implementation of Algebraic Specification*, ACM TOPLAS, Vol 15 Number 5, 1993

[Liskov&Scheifler 83]
B. Liskov, R. Scheifler, *Guardians and Actions: Linguistic Support for Robust, Distributed Programs*, ACM TOPLAS, July 1983

[Luckham 87]
D.C. Luckham, F.W. von Henke, B. Krieg-Brückner, *ANNA, A Language for Annotating Ada Programs*, LNCS vol 260, Springer, 1987

[Mahmood et al 84]
A. Mahmood, D.M. Andrews, E.J. McCluskey, *Executable Assertions and Flight Software*, AIAA/IEEE 6th Digital Avionics Systems Conference, pp 346-351, 1984

[Mañas&de Miguel 88]
J.A. Mañas, T. de Miguel, *From LOTOS to C*, K.J. Turner, editor, Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland), pp. 79-84, North-Holland, 1988

[Matsuoka&Yonezawa 93]
S. Matsuoka, A. Yonezawa, *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, Research Directions in Concurrent Object-Oriented Programming, G. Agha, P. Wegner, A. Yonezawa (Eds.), MIT Press, 1993

[van der Meulen 90]
E.A. van der Meulen, *Fine-Grain Incremental Implementation of Algebraic Specifications*, In Proceedings of the 2nd Intl. Conference on Algebraic Methodology and Software Technology AMAST'91, Springer, 1991

[Meyer 86]
B. Meyer, *Genericity versus Inheritance*, In Proceedings of the 1st OOPSLA, pp 391-405, Portland, Oregon, 1986

[Meyer 87]
B. Meyer, *Eiffel: A Language and Environment for Software Engineering*, Interactive Software Engineering, Inc, 1987

[Meyer 88]
B. Meyer, *Object-oriented Software Construction*, Prentice-Hall International (UK) Ltd, Hemel Hempstead, 1988

[Milner 80]
R. Milner, *A Calculus of Communicating Systems*, LNCS Vol 92, Springer Verlag, Berlin, 1980

[Moss 81]
J.E.B. Moss, *Nested Actions: an Approach to Reliable Distributed Computing*, PhD thesis, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[Mullender 93]
S. Mullender, *Interprocess Communication*, In "Distributed Systems", 2nd edition, S.Mullender (Ed.), Addison-Wesley, New York, 1993

[Murphy et al 89]
S.C. Murphy, P. Gunningberg, J.P.J. Kelly, *Implementing Protocols with Multiple Specifications: Experiences with Estelle, Lotos and SDL*, 9th IFIP WG 6.1 International Symposium on Protocol Specifications, Testing and Verification, Enschede, The Netherlands, June 1989

[Niemeyer&Peck 96]
P. Niemeyer, J. Peck, *Exploring Java*, O'Reilly and Associates inc., 1996.

[OMG 95]
Object Management Group, *The Common Object Request Broker: Architecture and Specification (Revision 2.0)*, Object Management Group, Framingham, Mass., 1995

[Padawitz 88]
P. Padawitz, *Computing in Horn Clause Theories*, EATCS Monographs on Theoretical Computer Science, Vol 16, Springer Verlag, Berlin, 1988

[Pereira et al 86]
L.M. Pereira, L. Monteiro, J. Cunha, and J.N. Aparício, *Delta-Prolog: A Distributed Backtracking Extension with Events*, Proceedings of the Third International Conference on Logic Programming, volume 255 Lecture Notes in Computer Science, Springer Verlag, London, July 1986

[Petri 62]
C.A. Petri, *Kommunikation mit Automaten*, Rhein, Westf. Inst. F. Instr. Math., Bonn, 1962

[Plat&Voss 95]
N. Plat, H. Voss, *The VDM++ Toolbox User Manual*, Technical Report, CAP Volmac & IFAD, ESPRIT-III project number 6500 "Afrodite", Doc Id: AFRO/CG/NP/VPPUM/V5.3, Sept 1995

[Plotkin 77]
G.D. Plotkin, *LCF Considered as a Programming Language*, Theoretical Computer Science, 5:223-255, 1977

[Pulli&Elmstrøm 93]
P. Pulli, R. Elmstrøm, *IPTES: A Concurrent Engineering Approach for Real-Time Software Development*, Real-Time Systems Journal, Vol 5, No 2/3, Kluwer Academic Publishers, Netherlands, May 1993

[Rabéjac 95]
C. Rabéjac, *Auto-surveillance logicielle pour applications critiques: méthodes et mécanismes*, PhD thesis, LAAS-CNRS, No 1095 (LAAS report No 95449), November 1995

[Reisig 85]
W. Reisig, *Petri Nets. An Introduction*. EATCS Monographs on Theoretical Computer Science, Vol. 4, Springer Verlag, 1985

[Reisig 86]
W. Reisig, *A Strong Part of Concurrency*, Revised proceedings of the 7th European Workshop on Application and Theory of Petri Nets, Oxford (UK), LNCS, June 1986

[Reisig 91]
W. Reisig, *Petri Nets and Algebraic Specifications*, Theoretical Computer Science 80, pp. 1-34, 1991

[Robinson 65]
J.A. Robinson, *A Machine-Oriented Logic Based on the Resolution Principle*, Journal of the Association for Computing Machinery, 12(1):23-41, 1965

[Roques 94]
C. Roques, *Modularité dans les spécifications algébriques: Théorie et applications*, Thèse de doctorat, No d'ordre 3308, Université Paris XI Orsay, 1994

[Rosenkrantz et al 78]
D.J. Rosenkrantz, R.E. Stearns, P.M. Lewis, *System Level Concurrency Control in Distributed Data Base*, ACM TODS, 3(2), pp 178-198, June 1978

[Rumbaugh et al 91]
J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991

[Rumbaugh 95]
J. Rumbaugh, *OMT: The Functional Model*, JOOP, March-April 1995

[Schnoebelen 88]
Ph. Schnoebelen, *Refined Compilation of Pattern-Matching for Functional Languages*, Science of Computer Programming, 11:133-159, 1988

[Shapiro 83]
E.Y. Shapiro, *A Subset of Concurrent Prolog and its Interpreter*, Technical Report, Weizmann Institute, Rehovot, February 1983

[Sheard 91]
T. Sheard, *Automatic Generation and Use of Abstract Structure Operators*, ACM TOPLAS, Vol 13, pp 531-557, 1991

[Schwartz&Mattern 92]
R. Schwartz, F. Mattern, *Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail*, Dept. of Computer Science, University of Kaiserslautern, Technical Report SFB124-15/92, Kaiserslautern, Germany, 1992, also available in "Distributed Computing", Vol. 7, No. 3, pp. 149-174, 1994

[Simons, Kwang&Mei 94]
A.J.H. Simons, L.E. Kwang, N.Y. Mei, *An Optimizing Delivery System for Object-Oriented Software,* in Object Oriented Systems, Vol I, pp 21-44, 1994

[Sinclair, Clynch&Stone 95]
D. Sinclair, G. Clynch, B. Stone, *An Object-Oriented Methodology from Requirements to Validation*, Procs. 2nd Intl. Conf. on Object-Oriented Information Systems, Springer, 1995

[Skeen 81]
D. Skeen, *NonBlocking Commit Protocols*, In Procs. ACM SIGMOD Intl. Conf. on Management of Data, pp. 133-142, ACM Press, 1981

[Sommerville 92]
I. Sommerville, *Software Engineering*, 4th revision, Addison Wesley, 1992

[Somogyi 87]
Z. Somogyi, *A System of Precise Modes for Logic Programs*, Procs. 4th Int. Conf. on Logic Programming, Melbourne, Australia, pp. 769-787, 1987

[Somogyi, Henderson&Conway 96]
Z. Somogyi, F. Henderson, T. Conway, *The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language*, Journal of Logic Programming, Elsevier, 1996.

[Spivey 89]
J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, London, 1989

[Srivastava 92]
A. Srivastava, *Unreachable Procedures in Object-Oriented Programming*, in LOPLAS, Vol I, pp 355-364, 1992

[Stallman&Sussman 77]
R. Stallman, G.J. Sussman, *Forward Reasoning and Dependency Directed Backtracking in a System for Computer-Aided Circuit Analysis*, Artificial Intelligence, Vol. 9, pp. 135-196, 1977

[Strohmeier 96]
A. Strohmeier, *Cycle de vie du logiciel*, In “Génie logiciel: principes, méthodes et techniques”, A. Strohmeier and D. Buchs (Eds.), Presses polytechniques et universitaires romandes, Lausanne, Switzerland, 1996

[Stroustrup 91]
B. Stroustrup, *The C++ Programming Language*, Second Edition, Addison Wesley, 1991

[Taubner 87]
D. Taubner, *On the Implementation of Petri Nets*, 8th European Workshop on Applications and Theory of Petri Nets, Saragosa, Spain, LNCS Vol 340, Springer Verlag, pp. 418-439, June 1987

[Tebra 87]
H. Tebra, *Optimistic And-Parallelism in Prolog*, In Procs. Parallel Architectures and Languages Europe PARLE’87, Eindhoven, The Netherlands, LNCS Vol. 259, June 1987

[Ueda 85]
K. Ueda, *Guarded Horn Clauses*, Technical Report TR-103, ICOT, June 1985

[Ungar&Smith Randall 87]
D. Ungar, B. Smith Randall, *Self: The Power of Simplicity,* SIGPLAN Notices 22,12, December 1987

[Verhofstad 78]
J.S.M. Verhofstad, *Recovery Techniques for Database Systems*, Computing Surveys, 10(2):167-195. June 1978

[Vonk 92]
R. Vonk, *Prototypage: l'utilisation efficace de la technologie CASE,* Masson & Prentice-Hall, Paris, 1992

[Walker, Floyd&Neves 90]
E. Walker, R. Floyd, P. Neves, *Asynchronous Remote Operation Execution in Distributed Systems*, Proc. IEEE 10th Int. Conf. on Distributed Computing Systems, May 1993

[Ward&Mellor 85]
P.T. Ward, S.J. Mellor, *Structured Development for Real-Time Systems*, volume 1-3, Yourdon Press, New York, 1985

[Wegner 87]
P. Wegner, *Dimensions of Object-Based Language Design*, in OOPSLA '87 Conference Proceedings, pp. 168-182

[Weihl 89]
W. Weihl, *Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types*, ACM Transactions on Programming Languages and Systems, Vol. 11, No 2, 1989

[Zaremski&Wing 95]
A.M. Zaremski, J.M. Wing, *Specification Matching of Software Components*, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, Oct. 1995

Appendix A.
Major Control Algorithms for CO-OPN

In this appendix are listed the most important algorithms for the control of the synchronizations. We use a syntax close to the Ada programming language, with some syntactic sugar and relaxed so as to allow copy-out parameters for functions and to let copy-in parameters be modified locally.

Appendix A.1 Some Relations Between Method Calls

Let us define the kinds of relations which exist between method calls: `Children`, `Dependent`, `After` and `LowerPriority`.

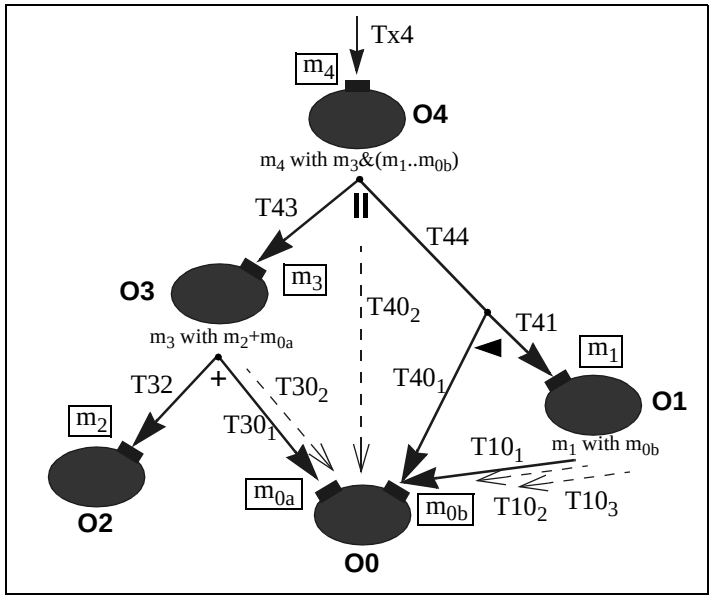


Figure 118. Example for the Terminology (with some `Stabilize` Requests for `O0`)

We have the following relationships in figure 118 with respect to Object O0:

- Children(Xid) are the subtransactions of Xid, i.e. the synchronizations and stabilizations which extend Xid downwards: Children(T44) = { T41, $T10_1$, $T10_2$, $T10_3$, $T40_1$}.
- After(Xid) are the transactions (of the events) which happen causally after Xid, due to a sequential operator or a stabilization: After($T10_1$) = { $T10_2$, $T10_3$, $T40_1$, $T40_2$ }.
- Dependent(Xid) are the transactions which are Children of Xid or come After Xid: Dependent($T10_1$) = Children($T10_1$) + After($T10_1$) = { $T10_2$, $T10_3$, $T40_1$, $T40_2$ }.
- First(List_of_Xids), where the Xids of List_of_Xids are connected by a sequence or a stabilization, returns the Xid corresponding to the event which comes first temporally: First(< $T10_1$,$T40_1$ >) = $T10_1$. If some Xids of List_of_Xids are also connected by a simultaneity, then First returns the list of Xids which come first in the respective thread of simultaneity: First(< $T30_1$,$T30_2$,$T10_1$,$T10_2$,$T10_3$,$T40_1$ >) = < $T30_1$, $T10_1$ >. We also have the following relation: List_of_Xids = First(List_of_Xids) + Dependent(First(List_of_Xids)).
- Independent(Xid) are the transactions which are connected to Xid by a simultaneity or an alternative: Independent($T30_1$) = { $T10_1$ } (relatively to O0).
- LowerPriority(Xid) are the transactions which are connected to Xid by a simultaneity or an alternative, i.e. the independent transactions, and which will be managed after Xid because of some priority. We have e.g. LowerPriority($T10_1$) = { $T30_1$ } because $T10_1$ comes from Object O1 which is lower in the hierarchy than O3 (and has therefore higher priority according to the semantics of CO-OPN).

Appendix A.2 Basic Synchronization Algorithms

There are three kinds of synchronization operators in CO-OPN: the sequence (“..”), the simultaneity (“&”) and the alternative (“+”). For each operator there is one algorithm for serving the corresponding request (ServeSerial, ServeSim, ServeAlt). The ServeSerial algorithm is also used for serving the first call of a synchronization as well as successive stabilization requests.

Function ServeSerial

ServeSerial simply evaluates the method and returns the status Success or Failure to the caller.

```
1  FUNCTION ServeSerial (CurrentXid: IN Xid;
2                        CurrentSync: IN SynchronizationVector;
3                        InitialState: IN ObjectState;
4                        ResultingState: OUT ObjectState;
5                        SuccessfulXids: OUT Set_of_Xids)
```

```
6  RETURN Status IS
7  BEGIN
8    IF EvalMethod(CurrentXid, CurrentSync,
                   InitialState, ResultingState,
                   SuccessfulXids)=Success THEN
9      RETURN Success;
10   ELSE
11     RETURN Failure;
12   END IF;
13 END ServeSerial;
```

Function ServeSim

ServeSim is called each time a new method call starts a simultaneous thread with some existing calls. It first evaluates the call in the appropriate context, without aborting any existing lower priority calls. This is possible since the different simultaneous threads are totally independent, both within the current Object and within sub-Objects being synchronized with, if any. If the evaluation succeeds, then all calls with lower priorities are aborted whether or not they really are in conflict with the current call (this is called *aggressive cancellation* in the field of computer simulation [Ferscha 96]). If the evaluation fails, then the function simply returns `Failure`.

In case of success, because of the optimistic strategy adopted for the evaluation of simultaneity, it is necessary to warn directly the emitter of the simultaneous synchronization in order to restart the calls with lower priority. If we naively send new answers to the lower priority calls after reevaluating them, then we will miss the ones which we have forgotten because they had initially failed, but which might succeed now, after the current call. We may also be unable to efficiently catch up with succeeded calls which are terminated in between and which are now under the control of the Object requesting the simultaneity.

It is important to notice that all simultaneous calls are completely serialized, the calls with highest priority being executed first.

```
1  FUNCTION ServeSim  (CurrentXid: IN Xid;
2                      CurrentSync: IN SynchronizationVector;
3                      StateBeforeSim: IN ObjectState;
4                      ObjRequestingSim: IN ObjectId;
5                      HigherPriority: IN List_of_Xids;
6                      LowerPriority: IN List_of_Xids;
7                      ResultingState: OUT ObjectState;
8                      SuccessfulXids: OUT Set_of_Xids)
9  RETURN Status IS
10 BEGIN
11   IF EvalMethod(CurrentXid, CurrentSync,
                   StateBeforeSim - Consumed(HigherPriority), ResultingState,
                   SuccessfulXids)=Success THEN
12     send(ObjRequestingSim,RestartLower,CurrentXid);
13     abort(LowerPriority);
14     RETURN Success;
```

```
15   ELSE
16     RETURN Failure;
17   END IF;
18 END ServeSim;
```

Comments on *ServeSim*

Line 11 tries to serve the new method call in the state the Object is supposed to have right after serving all simultaneous requests with higher priority. If it is successful, then the simultaneous requests with lower priority will have to be restarted, because the current one will probably have taken some of their tokens (line 12 for the global reset, and line 13 performs an optimization by anticipating locally the abort which will result from the global reset).

Function ServeAlt

ServeAlt is called at each arrival of a method call connected to an existing one by the means of an alternative operator. Its effect is to suspend the call and warn the emitter of the synchronization about it so that it is known that the call must be reactivated. This is because it is not possible for a single Object to evaluate several alternatives at the same time in the general case where there exist calls with lower priority than the alternatives: According to which of the alternatives should then the calls with lower priority be aborted or restarted ? In order to evaluate the alternatives in parallel instead of in sequence it would therefore be necessary to duplicate the whole synchronization sorrounding the alternative.

```
1  FUNCTION ServeAlt   (CurrentXid: IN Xid;
2                       ObjRequestingAlt: IN ObjectId)
3  RETURN Status IS
4  BEGIN
5     send(ObjRequestingAlt,Suspended,CurrentXid);
6  END ServeAlt;
```

The call identified by `CurrentXid` is now blocked until Object `ObjRequestingAlt` sends a `Restart` to the current Object with `CurrentXid` as argument.

Appendix A.3 The Stabilization Procedure

Introduction

The functions `StabilizeLowerObjects and Stabilize` are activated according to the semantics of CO-OPN at the end of each successful synchronization (rule BEH-SYNC) and in the middle of each sequence (rule BEH-SEQ).

The function `StabilizeLowerObjects` is called by every Object "`MySelf`" which successfully terminates a synchronization. The function is responsible for sending `stabilize` messages to all relevant Objects which are strictly lower than `MySelf`.

The function `Stabilize` is executed by each Object "`MySelf`" receiving the message `Stabilize`. The Object then starts a loop for firing all internal transitions until it has become stable. At each iteration it calls `StabilizeLowerObjects` in order to stabilize the lower Objects which successfully took part in the synchronization from the current transition, as well as the associated `StabClients` which are lower than `MySelf`. If `List_of_Xids` is the list of all transaction Xids which took part in the synchronization, then the function `DirectParticipants`(`List_of_Xids`) returns the list of lower Objects which took part in the first level of this synchronization.

Both functions use `CurrentXid` and `CurrentSync` as basis for building new subtransaction Xids and synchronization vectors, and return the set `AllSuccessfulXids` containing the new successful leaf transaction Xids they have created or inherited.

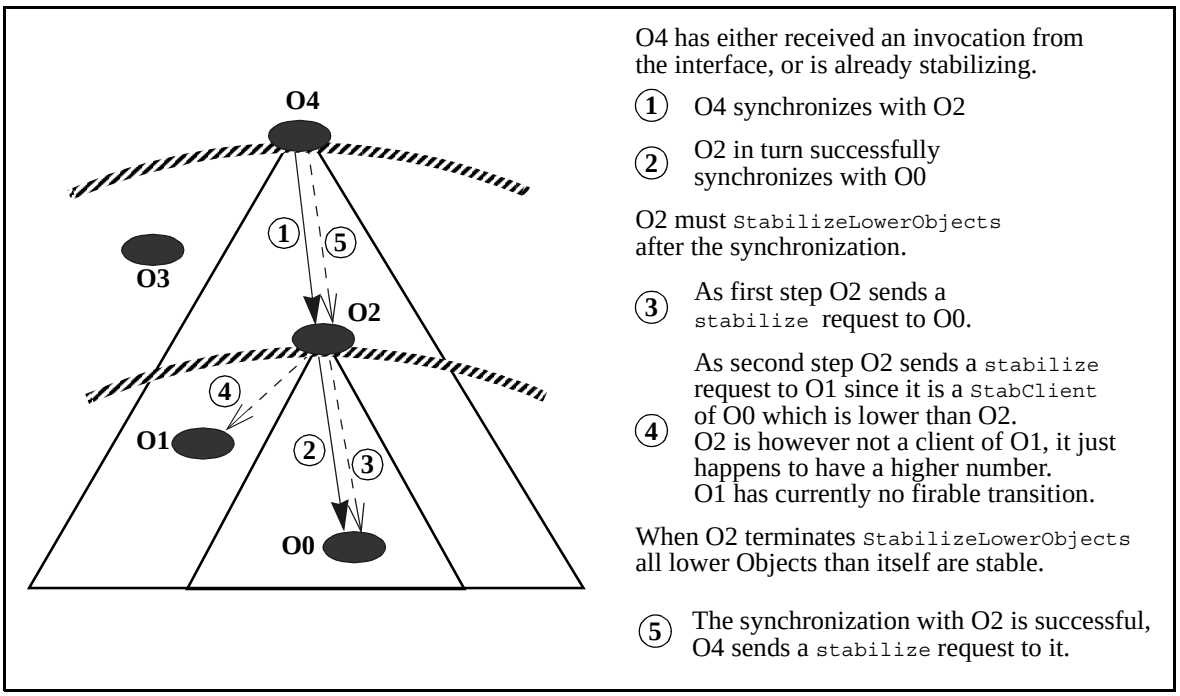


Figure 119. Mutual Dependency of `Stabilize` and `StabilizeLowerObjects`

The example of Figure 119 shows that `Stabilize` and `StabilizeLowerObjects` are mutually dependent for achieving correctly the work. We suppose here that `O4` is not client of `O3`, hence `O3` is outside of the big triangle with `O4` at the summit. Moreover Object `O3` is not a `StabClient` of any of the Objects within that triangle. That is why `O3` is never accessed

during the stabilization. This is the first advantage of our algorithm compared to an exhaustive stabilization scheme following blindly the total order of the Object dependency graph. The other advantage is that the Objects the state of which has been modified (such as O0 and O1) receive the command *stabilize* only once instead of one each time a new level of the synchronization they took part in terminates successfully (i.e. O0 and O1 are stabilized when the synchronization terminates in O2, but not again for O4). The algorithm actually works by layers, as shown in the above figure.

Function Stabilize

```
1  FUNCTION Stabilize (MySelf: IN ObjectId;
2                      CurrentXid: IN Xid;
3                      CurrentSync: IN SynchronizationVector;
4                      InitialState: IN ObjectState;
5                      ResultingState: OUT ObjectState)
6  RETURN Set_of_Xids IS
7    AllSuccessfulXids: Set_of_Xids := {};
8    InheritedXids: Set_of_Xids := {};
9    SubXid: Xid;
10   SubSync: SynchronizationVector;
11   Clock: natural := 1;
12 BEGIN
13   LOOP
14     SubXid := MakeLocalXidFrom(CurrentXid);
15     SubSync := MakeSyncFrom(CurrentSync,"stab"+Clock);
16     IF EvalAnyTransition(SubXid, SubSync,
                            InitialState, ResultingState,
                            SuccessfulXids)=failure THEN
17       delete(SubXid,SubSync);
18       RETURN FilterNonLeaf(AllSuccessfulXids);
19     END IF;
20     insert(AllSuccessfulXids,SuccessfulXids);
21     IF Card(Participants(SuccessfulXids)) > 1 THEN
22       InheritedXids := StabilizeLowerObjects(MySelf, SubXid, SubSync,
                                                DirectParticipants(SuccessfulXids));
23       insert(AllSuccessfulXids,InheritedXids);
24     END IF;
25     Clock := Clock+1;
26   END LOOP;
27 END Stabilize;
```

Comments on *Stabilize*

Lines 14 and 15 create a new Xid and synchronization vector for a local subtransaction which will encapsulate the next transition firing. This allows us to identify and abort, if necessary, all external stabilizations entailed by the firing of the internal transition. Lines 16 to 19 try to fire one more transition: If it fails then it means that the Object is stable, therefore we delete the unused subtransaction and return to the caller. When returning we keep only the leaf transaction Xids, since all intermediate Xids are contained within the latter, allowing us to prevent messages from carrying redundant information. Line 20 inserts the inherited subtransaction Xids into the set of successful Xids to be returned as result. Line 21 checks if there was a synchronization during the transition and if other Objects must be

stabilized before going on. Lines 22 and 23 care for the stabilization of all Objects, lower than `MySelf` and above or equal to the lowest of the `DirectParticipants`, which have become unstable during the successful subtransaction. These additional stabilization subtransactions are also remembered for the final result. Line 25 increments the local Clock to indicate the progress: This could have been avoided by nesting all the future events within subtransactions of the current transaction, but at the price of having to manipulate and transfer rapidly growing Xid structures. By creating a sibling transaction at each step of the stabilization, instead of a subtransaction (which would maybe better represent the causality), we obtain Xids whose length is bounded by O(*p*), *p* being the maximal depth of a synchronization in a connected Object dependency graph, instead of O(*f*), where *f* is the total number of methods and transitions firable on the path *p*. The number *f* is by hypothesis finite but may in practice be very high.

Function StabilizeLowerObjects

```
1  FUNCTION StabilizeLowerObjects (MySelf: IN ObjectId;
2                                  CurrentXid: IN Xid;
3                                  CurrentSync: IN SynchronizationVector;
4                                  ToStabilize: IN Set_of_ObjectId)
5  RETURN Set_of_Xids IS
6    AllSuccessfulXids: Set_of_Xids := {};
7    InheritedXids: Set_of_Xids := {};
8    LowestObject: ObjectId;
9    SubXid: Xid;
10   SubSync: SynchronizationVector;
11   Clock: natural := 1;
12 BEGIN
13   insert(ToStabilize,LowerThan(MySelf,StabClients(ToStabilize)));
14   WHILE ToStabilize /= {} LOOP
15     LowestObject := Lowest(ToStabilize);
16     SubXid := MakePartialXidFrom(CurrentXid);
17     SubSync := MakeSyncFrom(CurrentSync, <stab,Clock>);
18     send(LowestObject, SubXid, SubSync, stabilize);
19     wait(LowestObject, SubXid, stabilized, InheritedXids);
20     insert(AllSuccessfulXids,InheritedXids);
21     insert(ToStabilize,
              DirectParticipants(InheritedXids) +
                 ObjectsBetween(MySelf,
                                Lowest(DirectParticipants(InheritedXids)),
                                StabClients(Participants(InheritedXids))));
22     IF LowestObject = Lowest(ToStabilize) THEN
23        remove(ToStabilize,LowestObject)
24     END IF;
25     Clock := Clock+1;
26   END LOOP;
27   RETURN FilterNonLeaf(AllSuccessfulXids);
28 END StabilizeLowerObjects;
```

Comments on *StabilizeLowerObjects*
Line 13 adds to the set of Objects to stabilize all those which are `StabClients` of the unstable ones and which are lower than `MySelf` in the hierarchy. Line 14 ensures that the function does not return before there are no more Objects to stabilize. Line 15 selects as Object to stabilize the one which is the lowest in the set of unstable Objects. Lines 16 and 17 create a new Xid and synchronization vector for the subtransaction which encapsulates the `stabilize` request to be sent to the `LowestObject`. Lines 18 and 19 send `stabilize` to the LowestObject and waits for the result, a set of Xids executed by Objects which are now stable. Line 20 memorizes the set of inherited Xids for the future result. Line 21 adds to the set of Objects to be stabilized the ones which are `StabClients` of the newly modified Objects. By taking into account only those which are lower than `MySelf`, we can guarantee that the result `AllSuccessfulXids` only contains Xids related to stable Objects. It is then up to the caller to extend this result with the unstable Objects which are below itself. The test in lines 22 to 24 looks if the portion of the hierarchy which is stable is growing up. This portion is defined as being between the bottom and the Object which has just been stabilized. Line 25 increments the local time to mark the progress. Finally, line 27 returns the set of Xids corresponding to successful leaf transactions (i.e. we filter all Xids which are a subpart of a leaf transaction, since this information is redundant).

Proof of Completeness

We justify here the optimization of the stabilization algorithm by the restriction to "interesting" Objects. According to rules MONOTONICITY and STAB, any transition system $m_0 \xrightarrow{e} m_1$ can be rewritten as this:

$$m_0 + m(P) \ *\xrightarrow{e} \ m_1 + m(P)$$

where P is a maximal set of places which remain unchanged by e and its subsequent stabilization. If an Object o has all its places in P, then it can be excluded from e and its subsequent stabilization.

We must now prove that our algorithm really includes all Objects which are to participate in the stabilization.

Theorem 5: *The Stabilization is Maximal*

When `Stabilize` terminates for Object `MySelf`, it is guaranteed that:

1. `MySelf` and all lower Objects are stable.
2. No event has been overlooked during the stabilization.

Proof:

The implemented system has no autonomous activity: Any activity originates from an event coming from above through a method call. Each method call is enclosed in a synchronization, and each synchronization initiator calls `StabilizeLowerObjects` and applies `Stabilize` to itself before it terminates and in the middle of sequential synchronizations. Therefore method calls are either directly called from the system interface, or indirectly by synchronization from an internal transition. We will thus have to prove that:

1. Any firable transition belonging to an Object $o \sqsubset$ or equal to `MySelf` is fired before o accepts a new method call.
2. Any firable transition belonging to an Object $o \sqsubset$ `MySelf` is fired before `StabilizeLowerObjects` returns.

Sub-proof #1

Within a given transaction tree, there are two situations where an Object o which has already served a method call mc_1 may receive a new method call mc_2. The first is when the new method call mc_2 is simultaneous or alternative w.r.t. the already terminated method call mc_1, in which case mc_2 is to be executed in another context than mc_1, and therefore the context of mc_1 does not need to be stabilized before mc_2 is served. The other situation is when mc_2 is to be executed in the context resulting from mc_1 because it logically succeeds to mc_1. There is a single Object, the originator of the synchronization containing mc_1, which might produce such a situation. But this Object is responsible for calling `StabilizeLowerObjects` before it can continue with any other activity. Moreover, `StabilizeLowerObjects` systematically proceeds from the lowest Objects and upwards. Therefore Object o will always receive the `stabilize` command before any method call mc_2.

Sub-proof #2

There are two ways for an Object o to become unstable. The first is by a trivial method call, in which case the complete stabilization is performed right after. The second is when o has an internal transition t which synchronizes with a method of another Object o_2 ($o_2 \sqsubset$ or equal to o, and $o \sqsubset$ or equal to `MySelf`), the state of which has changed, allowing t to be fired. In other words, o is a `StabClient` of o_2 and has become unstable by virtue of CO-OPN's anti-inheritance of instability. Since the state of o_2 has changed, Object `MySelf` will also finally take care of Object o because o is a `StabClient` of o_2, as described by function `StabilizeLowerObjects` (line 21). ◊

Appendix A.4 Internal Structure for Managing Synchronizations

The following figure, which corresponds to Object `O0` in the situation of figure 98 on page 164, where `T10` is terminated, `T40` running, and `T30` not yet arrived, gives an idea of what the internal data structures could look like. The ovals represent copies of Objects in different states.

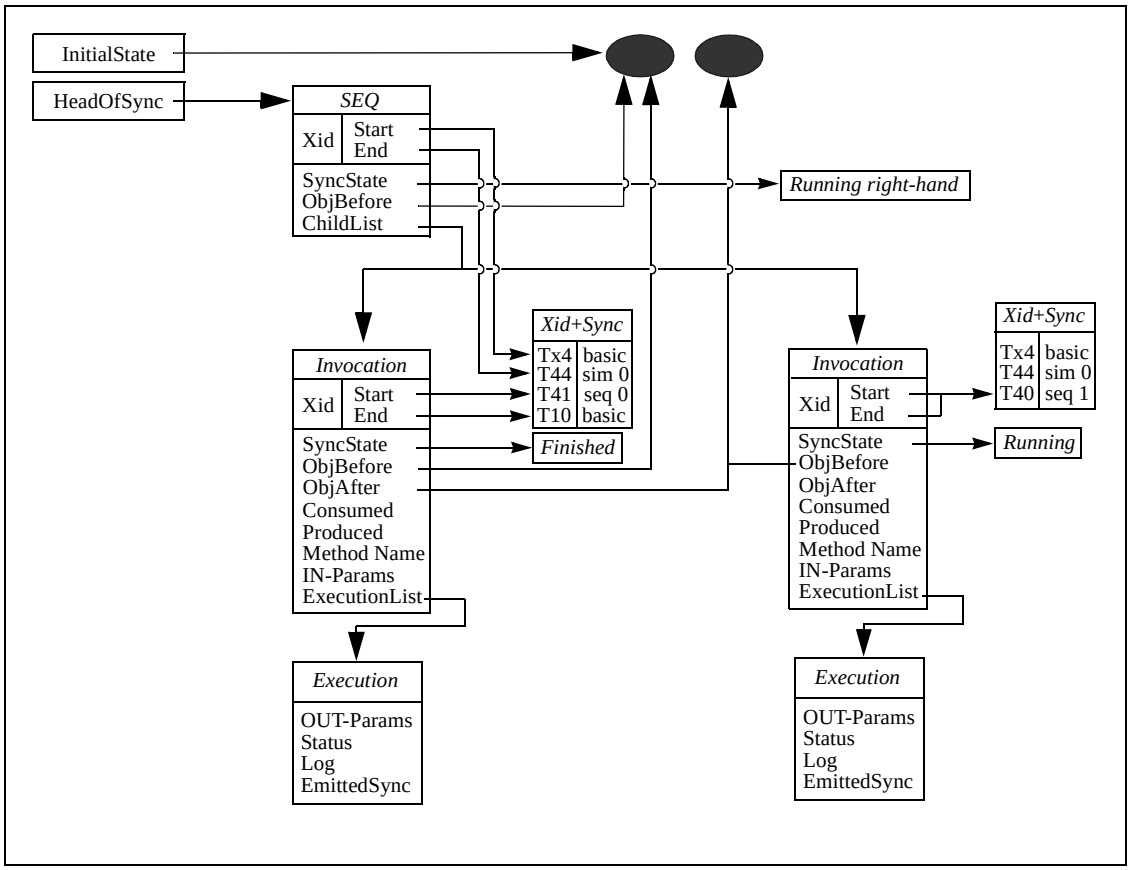


Figure 120. Internal Structure for Managing the Synchronizations (without Stabilization)

Consecutive Object states can be seen as being stacked up, the top of the stack being pointed to by the last invocation of the synchronization, and the bottom being indicated by the global variable `InitialState`. Here is an explanation of the different structures:

1. The global variables `InitialState` and `HeadOfSync` point respectively to the first Object state saved and the root of the current synchronization tree.

2. The *Seq* structure corresponds to the data needed for managing sequential synchronizations. Analog structures are instantiated for the other kinds of synchronizations.
   - `Xid` points to the range of the Xid and synchronization vector which we are concerned with. This helps comparing Xids efficiently in order to calculate the context of incoming calls.
   - `SyncState` indicates the state of the current sub-part of the synchronization, i.e. whether it is running, finished, aborted etc.
   - `ObjBefore` points to a copy of the local Object which corresponds to the state it had before the given sub-part of the synchronization.
   - `ChildList` is the list of child transactions of the sequence.

3. The *Invocation* structure contains the data needed for managing incoming method calls.
   - `ObjBefore` points to a copy of the local Object which corresponds to the state it has after the given method call.
   - `Consumed` refers to an array where each entry contains the tokens consumed in a given place.
   - `Produced` refers to an array where each entry contains the tokens produced in a given place.
   - `MethodName` indicates the application-level procedure to execute for the given method call (with the dispatching needed for choosing between the automatically generated or the hand-written versions).
   - `In-Params` points to a structure where the input parameters of the incoming method are saved.
   - `ExecutionList` is the list of *Execution* structures corresponding to the different firings of the given method.

   For internal transitions there exists an analog structure, which does not have the `In-Params` field.

4. The *Execution* structure contains the data corresponding to each single execution of the method or transition it is connected to.
   - `OUT-Params` points to a structure where the output parameters of the incoming method are saved.
   - `Status` indicates whether the execution succeeded or failed.
   - The `Log` contains all the information for undoing operations and in particular it serves as stack for the choice points created during the execution of the method or transition.
   - `EmittedSync` points to the data needed for managing outgoing synchroniza-

tions. Its role is similar to the global variable HeadOfSync at the level of method and transition executions.

When the event is an internal transition, the Execution structure does of course not have the `OUT-Params` field.

Appendix B.
Messages Supported by the Control Layers

In this appendix we give the list of messages exchanged between the different Object levels. We do not give all the arguments, only the ones which are necessary for their primary intention. For instance, the timestamp parameter is not shown.

1. The messages managed on behalf of the controlled action class:
   - There is one implementation-level `m(ObjectId,Xid,SynchronizationVector,InArgumentList)` for each method `m` of the specification exported by Object `ObjectId`. The caller must of course provide the input arguments (`InArgumentList`) to the called method in addition to the Xid and the sychronization vector of the subtransaction enclosing the call.
   - `Reply_m(ObjectId,Xid,OutArgumentList)`, the reply to Object `ObjectId` of the specified method `m` which was called within transaction `Xid`. This is a high-level view where each `m` has its own reply method in order to transmit the success of a synchronization, as well as the resulting `OUT` parameters. In a lower-level approach, there would be a unique remote procedure, called `success`, with an untyped argument to be unpacked according to the profile of the method `m` concerned.
   - `Stabilize(ObjectId,Xid,SynchronizationVector)` tells (within subtransaction `Xid`) Object `ObjectId` to stabilize itself and to care for the stabilization of all the Objects it requests services from. Object `ObjectId` replies `Stabilized(Xid)` when terminated.
   - `Abort`(`ObjectId`,`Xid`, `Failure` | `Kill` | `Exception`). The reason `Failure` corresponds to the reply where a method failed because of a lack of resources for instance. The `Exception` corresponds to any exception raised during the remote execution of a request; ideally an additional string should give more information to the caller. The `Kill` does not correspond to a reply: It is transmitted from above and downwards, and corresponds to all the cases where a search is abruptly terminated by a caller. If an `Abort` message is received for a `Xid` which is already aborted, it is simply ignored.
2. Additional messages for the resolution layer:
   - `Retry(ObjectId,Xid)` for requesting from Object `ObjectId` a new answer for the non-deterministic method called initially within subtransaction `Xid`.

All dependents of `Xid` are aborted.

- `Reset(ObjectId,Xid)` for telling Object `ObjectId` to reinitialize and suspend the subtransaction `Xid`. All dependents of `Xid` are aborted, and `Xid` itself must be explicitly restarted. To *reinitialize* means that the search will start from scratch (by opposition to a `Retry` request). The requesting Object does not know the new Xids within which the `Reset` will take effect at Object `ObjectId`.
- `Restart(ObjectId,Xid)` for telling Object `ObjectId` to immediately reexecute `Xid` after reinitializing it. All dependents of `Xid` are aborted. The requesting Object does not know the new Xids within which the `reset` will take effect at Object `ObjectId`.
- `RestartLower(ObjectId,Xid)` for telling Object `ObjectId` to immediately `Restart` all Xids with strictly lower priority than `Xid`. All their dependents are aborted. The requesting Object does not know the new Xids within which the `RestartLower` will take effect at Object `ObjectId`.
- `RestartDependent(ObjectId,Xid)` tells Object `ObjectId` to immediately `Restart` the Xid(s) dependent on `Xid`. The requesting Object does not know the new Xids within which the `RestartDependent` will take effect at Object `ObjectId`.
- `Suspended(ObjectId,Xid)` informs Object `ObjectId` that `Xid` has been suspended and will remain so until `ObjectId` sends to it a `Restart`.

3. Additional messages for the concurrency control and lower layers:
   - Messages for the *two phase commitment* protocol (`RequestVote`,`Vote` (with argument `Yes` or `No`), `Decision` (with argument `Commit` or `Abort`))
   - `QueryStability (ObjectId)` and the answers (`Unstable`/`Stable`). This reports about the *strong stability* of an Object `ObjectId`, i.e. whether it has terminated (and committed) its current activity and no synchronization or stabilization request is pending in its input queue.
   - `Terminate (ObjectId)` tells Object `ObjectId` to immediately cease all activity and exit. This generally supposes that `ObjectId` has been beforehand progressively prepared to terminate, although it is also useful to be able to stop a distributed prototype unconditionally.
   - `PrepareTermination (ObjectId)` and the answer `ReadyForTermination` are exchanged at the level of the prototype interfaces to the asynchronous world. The purpose is to block the system for any new input while still letting the current activities terminate normally. When all the interfaces have replied `ReadyForTermination` it means that the prototype is blocked and may be halted by sending the command `Terminate` to all Objects.
   - additional queries for ensuring that a given node has not crashed, which we do not detail here.

Appendix C.
Compilation of CO-OPN Objects

Appendix C.1 Semantics of the Source Language

As for AADTs, the semantics of the source language defined for Objects is a restriction of CO-OPN: We will apply rewriting for the algebraic terms of Objects, and resolution for their dynamic behaviour. As explained in chapter 4, rewriting is an interpretation of conditional equations which limits the semantics of algebraic specifications by orienting the equations. This allows more efficient implementations. For the Object part, we proceed similarly by imposing strong modes for the parameters of CO-OPN methods, which has the advantage of making explicit the direction of the data flow, and by establishing a well defined order in the resolution process, which helps deciding rapidly whether an event is firable or not.

Whereas chapter 6 describes the distributed resolution mechanism, we will concentrate here on the local view of this mechanism, as seen from inside a transition or method.

The following definition of the source language is a continuation of section 4.3 on page 95. For every method of a CO-OPN specification the compiler requires a *unique* and *complete* mode declaration; this allows us to determine by static analysis the data flow within a behavioural axiom. The data flow is interpreted so that all variables are assigned exactly once. It is clear that the meaning of the mode declaration is not the same for the caller as for the callee: The interpretation changes according to whether the variables constitute the formal or the actual parameters of a method. Let us define a notion of single-assignment variable.

Definition 42: *Binding of a Variable*

Let be an Object module $OM = < \Omega, P, X, \Psi >$, a behavioural axiom $(Cond \Rightarrow Event : Pre \rightarrow Post) \in \Psi$. A *binding* of a variable $x \in X$ is a function $Binding: X \rightarrow \{ free, bound \}$ defined as follows:

1. The initial binding of x is *free*;

2. The binding of *x* becomes *bound* the first time it is written to;
3. If the binding of *x* is *bound*, it cannot be written to, neither its binding become *free* again.

If the binding of *x* is *free* then *x* cannot be read from. ◊

Definition 43: *Mode and Boundedness of Variables*

Let be a method *m* defined by or synchronized with in a behavioural axiom *ax* of a CO-OPN specification, then:

- *TermsIn*(*m*) is the tuple of parameters of mode `IN` of *m* in *ax*;
- *TermsOut*(*m*) is the tuple of parameters of mode `OUT` of *m* in *ax*;
- *VarsIn*(*m*) is the tuple of variables of *TermsIn*(*m*) in *ax*;
- *VarsOut*(*m*) is the tuple of variables of *TermsOut*(*m*) in *ax*;
- *FreeVarsIn*(*m*) is the subset of *VarsIn*(*m*) of variables which are free when used as argument for *m*;
- *FreeVarsOut*(*m*) is the subset of *VarsOut*(*m*) of variables which are free when used as argument for *m*;
- *BoundVarsIn*(*m*) is the subset of *VarsIn*(*m*) of variables which are bound when used as argument for *m*;
- *BoundVarsOut*(*m*) is the subset of *VarsOut*(*m*) of variables which are bound when used as argument for *m*;

◊

We extend the domain of these functions to general synchronization expressions. The results are then tuples, the arity of which is equal to the sum of the arities of the elementar methods composing the synchronization expression. No specific order is assumed.

The following definition gives the syntactic criteria which characterize a *well-formed* method or transition definition. Recall from chapter 3 that given a signature $\Sigma=<S,F>$ where $F = C \cup OP$ (respectively the set of *constructors* and *defined operations* of *F*), $T_{C,X}$ denotes the set of all terms built over constructor symbols in *C* and variables in *X*.

Definition 44: *Well-Formed Method or Transition Definition*

Let *Spec* be a CO-OPN specification and *OM* an Object module of *Spec*. An event $e \in \mathbf{E}_{(T_{\Sigma,X}),M(Spec),O}$ is defined by a set *Beh-Rules*(*e*) of behavioural axioms in *OM*. The definition of *e* is *well-formed* if for all (*Cond* ⇒ *e* `With` *Sync* : *Pre* → *Post*) ∈ *Beh-Rules*(*e*), $e_1,e_2 \in \mathbf{E}_{(T_{\Sigma,X}),M(Spec),O}$:

1. *TermsIn*(*e*) and *TermsOut*(*Sync*) are tuples of linear terms in $T_{C,X}$;
2. $Pre=(Pre_p)_{p\in P}$ is a *P*-indexed family of linear terms in $T_{[C],X}$;
3. *TermsOut*(*e*), *TermsIn*(*Sync*), $Cond \in T_{F,X}$;
4. $Post=(Post_p)_{p\in P}$ is a *P*-indexed family of terms in $T_{[F],X}$;
5. $VarsOut(e) \cup VarsIn(Sync) \cup Vars(Cond) \cup Vars(Post) \subseteq VarsIn(e) \cup Vars(Pre)$ *VarsOut*(*Sync*);
6. For all e_1 & e_2 of *Sync*,
   - $FreeVarsOut(e_1) \cap FreeVarsOut(e_2) = \varnothing$
   - $FreeVarsOut(e_1) \cap VarsIn(e_2) = \varnothing$
   - $VarsIn(e_1) \cap FreeVarsOut(e_2) = \varnothing$.
7. For all e_1+e_2 of *Sync*, and for all equation $c \in Cond$,
   - $Vars(Post) \cap FreeVarsOut(e_1+e_2) \subseteq FreeVarsOut(e_1) \cap FreeVarsOut(e_2)$
   - $Vars(c) \cap FreeVarsOut(e_1+e_2) \subseteq FreeVarsOut(e_1) \cap FreeVarsOut(e_2)$
   - $FreeVarsOut(e_1) \cap VarsIn(e_2) = \varnothing$
   - $VarsIn(e_1) \cap FreeVarsOut(e_2) = \varnothing$.

◊

All linearity constraints are imposed for simplifying the compilation of pattern-matching by reusing Schnoebelen's algorithm as described in section 4.8 on page 107. The same reason justifies the requirement of terms in normal form, i.e. terms in $T_{C,X}$. Rules 6 and 7 are designed to verify the correctness of the data flow in presence of simultaneities and alternatives, i.e. that no implicit conditions arise by the sharing of variables between independent activities. Let us now define the rewriting of multi-sets when considered as simple lists (the numbering of the source language rules *Sn* continues from chapter 4):

Definition 45: *Application 'Rewr' on Multi-Sets of Terms*

Rewr: $T_{[F]} \rightarrow T_{[C]} \cup \{\texttt{error}\}$ is defined by:

$$\text{(S9)}\quad \frac{Rewr[t_i] = \texttt{error},\ t_i \in \{t_1,...,t_n\}}{Rewr[\{t_1,...,t_n\}] = \texttt{error}}$$

$$\text{(S10)}\quad \frac{\forall\ i = 1...n,\ Rewr[t_i] \neq \texttt{error}}{Rewr[\{t_1,...,t_n\}] = \{Rewr[t_1],...,Rewr[t_n]\}}$$

◊

We also need to define rewriting on events, because of the algebraic expressions occurring within synchronizations:

Definition 46: *Application 'Rewr' on Events*

Let *Spec* be a CO-OPN specification. For all $e,e_1,e_2 \in \mathbf{E}_{T_F,M(Spec),O}$, *Rewr*: $[\mathbf{E}_{T_F,M(Spec),O}] \rightarrow [\mathbf{E}_{T_C,M(Spec),O}] \cup \{\texttt{error}\}$ is defined by:

$$\text{(S11)}\ \frac{Rewr[t_i] = \texttt{error},\ t_i \in \{\ t_1,...,t_n\}}{Rewr[e(t_1,...,t_n)] = \texttt{error}}$$

$$\text{(S12)}\ \frac{\forall\ i = 1...n,\ Rewr[t_i] \neq \texttt{error}}{Rewr[e(t_1,...,t_n)] = e(Rewr[t_1],...,Rewr[t_n])}$$

$$\text{(S13)}\ \frac{Rewr[e_i] = \texttt{error},\ e_i \in \{e_1,e_2\}}{Rewr[e_1\ \texttt{With}\ e_2] = \texttt{error}} \qquad \text{(S14)}\ \frac{\forall\ i = 1...2,\ Rewr[e_i] \neq \texttt{error}}{Rewr[e_1\ \texttt{With}\ e_2] = Rewr[e_1]\ \texttt{With}\ Rewr[e_n]}$$

$$\text{(S15)}\ \frac{Rewr[e_i] = \texttt{error},\ e_i \in \{e_1,e_2\}}{Rewr[e_1\ ..\ e_2] = \texttt{error}} \qquad \text{(S16)}\ \frac{\forall\ i = 1...2,\ Rewr[e_i] \neq \texttt{error}}{Rewr[e_1\ ..\ e_2] = Rewr[e_1]\ ..\ Rewr[e_n]}$$

$$\text{(S17)}\ \frac{Rewr[e_i] = \texttt{error},\ e_i \in \{e_1,e_2\}}{Rewr[e_1\ \&\ e_2] = \texttt{error}} \qquad \text{(S18)}\ \frac{\forall\ i = 1...2,\ Rewr[e_i] \neq \texttt{error}}{Rewr[e_1\ \&\ e_2] = Rewr[e_1]\ \&\ Rewr[e_n]}$$

$$\text{(S19)}\ \frac{Rewr[e_i] = \texttt{error},\ e_i \in \{e_1,e_2\}}{Rewr[e_1 + e_2] = \texttt{error}} \qquad \text{(S20)}\ \frac{\forall\ i = 1...2,\ Rewr[e_i] \neq \texttt{error}}{Rewr[e_1 + e_2] = Rewr[e_1] + Rewr[e_n]}$$

◊

The following is a set of basic predicates needed for the definition of the partial semantics *Resolve-PSem* (definition 48 below) in the frame of the resolution calculus. In the case where a non-deterministic operation fails, the underlying mechanism always starts backtracking. This should not be confused with the situation where a rewriting step returns `error`: The whole computation will then stop, unless the developer defines an adequate exception handler.

Definition 47: *Basic Predicates*

Let *Spec* be a CO-OPN specification, $Spec^A = < \Sigma,X,\Phi >$ its associated algebraic specification, $A = T_{(\Sigma,X)}/\Phi$ the initial model of $Spec^A$, and the Object module $OM = < \Omega, P, X, \Psi >$ of *Spec*. $\forall\ m,m_1,m_2 \in Mark_{Spec,A}$, $event,e \in \mathbf{E}_{T_F,M(Spec),O}$ and grounding substitution $\sigma \in \sigma_g$:

ChooseAxiom(IN, OUT) is a non-deterministic predicate which chooses one among the behavioural axioms of event *e*:

$$\text{(S21)}\quad \frac{\textit{Beh-Rules}(e) = \{(\textit{Cond}_i \Rightarrow \textit{Event}_i : \textit{Pre}_i \rightarrow \textit{Post}_i)_{i=1..m}\}}{\textit{ChooseAxiom}(e, \textit{Cond}_1 \Rightarrow \textit{Event}_1 : \textit{Pre}_1 \rightarrow \textit{Post}_1)}$$

Match(IN, IN, OUT) returns the substitution σ such that *event* matches *e*. In particular, if *e* is a method, then it means that all actual IN parameters of *event* satisfy the conditions expressed by the corresponding formal parameters of *e*:

$$\text{(S22)}\quad \frac{\sigma e = \textit{event}}{\textit{Match}(e, \textit{event}, \sigma)}$$

ApplyPre(IN, IN, OUT, OUT) finds input tokens in *m* which satisfy the preconditions $\{t_1,...,t_n\}$ and returns the resulting state and substitution:

$$\text{(S23)}\quad \frac{\sigma\{t_1,...,t_n\} \subseteq m}{\textit{ApplyPre}(m, \{t_1,...,t_n\}, m\text{-}\sigma\{t_1,...,t_n\}, \sigma)}$$

ApplySync(IN, IN, OUT, OUT) performs the given synchronization *e* according to the inference rules of CO-OPN and returns the resulting state and substitution:

$$\text{(S24)}\quad \frac{m_1 \overset{\sigma e}{*\!\!\rightarrow} m_2}{\textit{ApplySync}(m_1, e, m_2, \sigma)}$$

ApplyPost(IN, IN, OUT) calculates the postcondition $\{t_1,...,t_n\}$ from a given marking *m* and returns the resulting state:

$$\text{(S25)}\quad \frac{}{\textit{ApplyPost}(m, \{t_1,...,t_n\}, m+\{t_1,...,t_n\})}$$

◊

We are in fact getting closer and closer to a Prolog-like view of execution, since the textual order is now part of the semantics, and is enforced by the incremental computation of substitutions. The expression $\sigma_1\sigma_0$, denotes the composition of the substitutions σ_1 and σ_0. Since we only work with grounding substitutions, the domains and codomains of σ_1 and σ_0 are disjoint, which means that their composition is simply a union.

Definition 48: *Resolve-PSem*

Let *Spec* be a well-formed CO-OPN specification, $Spec^A = < \Sigma, X, \Phi >$ its associated algebraic specification, $A = T_{(\Sigma,X)}/\Phi$ the initial model of $Spec^A$, and the Object module $OM = < \Omega, P, X, \Psi >$ of *Spec*. The semantics $Resolve\text{-}PSem_A(OM)$ is the couple made of a transition system and a grounding substitution $< < Mark_{Spec,A} \times \mathbf{E}_{A,M(Spec),O} \times Mark_{Spec,A} >, \sigma_g >$ obtained by the given ordered evaluation of tests and basic predicates. $\forall m_0, m_1, m_2, m_3 \in Mark_{Spec,A}$, $e, event \in \mathbf{E}_{A,M(Spec),O}$, and grounding substitutions $\sigma_0, \sigma_1, \sigma_2 \in \sigma_g$:

$Resolve\text{-}PSem(m_0, event, m_3, \sigma_2\sigma_1\sigma_0) =$
$\quad ChooseAxiom(event, Cond \Rightarrow e \texttt{ With } Sync : Pre \rightarrow Post) \wedge$
$\quad Match(e, event, \sigma_0) \wedge$
$\quad ApplyPre(m_0, Rewr[\sigma_0 Pre], m_1, \sigma_1) \wedge$
$\quad ApplySync(m_1, Rewr[\sigma_1\sigma_0 Sync], m_2, \sigma_2) \wedge$
$\quad Rewr[\sigma_2\sigma_1\sigma_0 Cond] = \texttt{true} \wedge$
$\quad ApplyPost(m_2, Rewr[\sigma_2\sigma_1\sigma_0 Post], m_3)$

The modes of *Resolve-PSem*'s arguments are respectively IN,IN,OUT,OUT. ◊

This kind of ordered resolution strategy is equivalent to *SLD-Resolution* [Robinson 65]. A well-known and important property of SLD-Resolution w.r.t. deduction is that it is *sound* (all solutions obtained are also solutions of the original problem), but *not complete* since the operational mechanism proceeds by depth first exploration of all solutions, meaning that an infinite loop may block the search even though a correct answer exists.

Appendix C.2 Semantics of the Target Language

The source language of the preceding section is an intermediate language designed to have identical semantics with the target language. The difference lies in the amount of operational details: The target language has lower-level primitives so that it is possible to obtain better performance by optimizing the scheduling of instructions. Another notable change is that the state of the Object is no longer given as explicit argument to the primitives.

In order to define the target language for the Object part of CO-OPN we will need new constructs, which are no longer purely functional, compared to the Adt part of CO-OPN.

Definition 49: *Object Target Language*

Starting from definition 31 on page 98, there are four additional predefined types:

- *Place* is the type of all places, regardless of the type of their contents.
- *AxiomName* is an enumerated type defined for each method and transition and is used for distinguishing between the behavioural axioms.
- *TXid* is the root type of all method invocation identifiers. This type receives a new subtype for each method of the specification.
- *MessageId* allows identifying all messages or internal events relative to a method invocation.

The predefined functions, predicates and control structures are:

- `Get`(*p*, *x*) is a non-deterministic predicate which removes any token from *p* (if *p* is not empty , otherwise `Get` fails) and assigns its value to the variable *x*.
- `Put`(*p*, *Te*) inserts a token of value *Te* into place *p*.
- `DeclareSync`(*Sync*(m_1,...,m_n)) declares to the concurrency control layer the correspondence between the method invocation identifiers (to be stored in the variables m_1,...,m_n of type *TXid*) and the forthcoming synchronization *Sync* (itself of type *TXid*). The functions `seq`, `sim` and `alt` are available for describing the structure of *Sync*.
- `ChooseAxiom`({$axiom_1$,...,$axiom_n$}, *ChosenAxiom*) is a non-deterministic predicate which assigns to the variable *ChosenAxiom* one among the given list of axioms. `ChooseAxiom` fails if the list of axioms is empty.
- `Send`(*m*, Te_1, ..., Te_n) starts a method invocation identified by *m* (of a type which is itself a subtype of *TXid*) by sending a request with a complete list of actual `IN` parameters Te_1, ..., Te_n.
- `Receive`(*m*, x_1, ..., x_r) delivers the successful reply to the method invocation `Send`(*m*,...) and assigns the variables x_1, ..., x_r to be used as the respective `OUT` parameters of the called method.
- `Kill`(*m*) aborts the method invocation identified by *m*. It is not an error to `Kill` a failed or an already aborted invocation.
- `NextEvent` waits for and returns to its caller the *MessageId* of the next incoming message.
- `Reply`(*m*) is the *MessageId* of the (expected or already received) reply to an invocation started with `Send`(*m*, ...). The reply may consist in a `Success`(*m*) or a `Failure`(*m*). A failed method invocation does not result in a visible action at the application layer unless it is necessary for managing alternatives: That is when a branch of an alternative fails irremediably and requires the application code to launch one of the remaining alternatives. In all other cases failed invocations directly produce invisible backtracks.
- The function "`>=`" returns a meta-boolean value which tells if the left-hand side *MessageId* argument is causally dependent on (i.e. in sequence with) the right-hand side *MessageId* argument. The answer is directly deduced from

the structure of the specified synchronization expression.

- The function "`=`" returns the meta-boolean value `true` if its arguments of type *MessageId* designate the same message, otherwise it returns `false`.
- `Done`(*m*) where *m* is an invocation identifier returns a meta-boolean value which tells if the corresponding `Receive`(*m*,...) has been performed.
- `Repeat` *Instructions* `Until` *B* has the usual procedural semantics; *B* is a meta-boolean expression.
- The control structure "`If` *B* `Then` *Instructions* `EndIf`" without an `Else` part has been added since we are no longer only compiling into expressions: `If` *B* `Then Fail Endif` is for instance a frequent construct. *B* is a meta-boolean expression.
- `Fail` immediately aborts the current procedure call and indicates to the caller that the search is interrupted.

◊

The purpose of defining a subtype of *TXid* for each method *m* of the specification is to allow the automatic generation of strongly and statically typed `Send` and `Receive` primitives. This assumes that the target programming language, such as Ada95, permits overloading. A less elegant alternative is to have a single *TXid* type to identify the invocations, and to define new primitives `Send_m` and `Receive_m` for each method *m*. The same reasoning is applied to the type *Place* in order to obtain strongly and statically typed `Get` and `Put` primitives.

Since we do not make use of any concurrency related construct at the application level, we do not need any new features from the object-oriented final programming language. The list given in section 4.4.1 on page 101 is therefore also valid for implementing the Object part of CO-OPN.

Appendix C.3 Compilation of Behavioural Axioms

This appendix describes the major algorithms involved in the compilation of behavioural axioms. For the sake of brevity and legibility we do not list here the optimized versions.

Compilation of Variable Accesses

Thanks to the mode declarations, it is possible to determine statically how to compile accesses to variables. We give below the corresponding compilation rules. A similar strategy may be found in [Somogyi 87]: The general case of Prolog requires however unification instead of simple pattern-matching.

Definition 50: *Rules for the Compilation of Variable Accesses*

Let be an Object module $OM = < \Omega, P, X, \Psi >$, a behavioural axiom ($Cond \Rightarrow e$ **WITH** $Sync : Pre \rightarrow Post$) $\in \Psi$. A variable $x \in X$ is a *formal parameter* when it occurs in e and an *actual parameter* when it occurs in *Cond*, *Sync*, *Pre* or *Post*.

Compilation rules for formal parameter *x*:

Boundedness \ Mode	IN	OUT
free	pattern-matching and assignment	error: parameter not assigned
bound	pattern-matching and equality test	standard read access

Compilation rules for actual parameter *x*:

Boundedness \ Mode	IN	OUT
free	error: parameter not assigned	pattern-matching and assignment
bound	standard read access	pattern-matching and equality test

Cond and *Post* have by definition all their parameters of mode IN, whereas *Pre* is considered as having only OUT parameters. ◊

The cases requiring pattern-matching followed by an equality test correspond to the allowed uses of non-linearity. Auxiliary variables (akin to variable renamings in Prolog implementations) are needed here for the correct verification of the implicit equality. For instance, the method call $m(c_1(x),c_2(x,y))$ with actual parameters of modes OUT and OUT is compiled into Ada like this:

```
Send(m);
...
Receive(m,p1,p2);
c1_match(p1,x,matched);
IF NOT matched THEN RETURN; END IF;
c2_match(p2,x1,y,matched);
IF NOT matched THEN RETURN; END IF;
IF x/=x1 THEN RETURN; END IF;
```

The variable *x* is considered as *free* in the first term $c_1(x)$ and as *bound* in the second term $c_2(x,y)$. The functions `c1_match` and `c2_match` are generated according to the description of section 4.8 on page 107.

For reasons of legibility, we have omitted the management of variable naming in all following algorithms: It is assumed that the variables are adequately declared by a lower-level mechanism. The generated code for the synchronizations is not optimized for the same reason, leading to redundant tests at prototype run-time. To eliminate these useless tests, a more advanced lookahead mechanism must be implemented in the compilation algorithms.

Function GenBehAxiomList

The outermost compilation algorithm is `GenBehAxiomList` which generates the code for a method or transition defined by a set `BehAxioms` of behavioural axioms.

```
1  FUNCTION GenBehAxiomList(BehAxioms: IN List_of_BehAxioms)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4    Axiom : BehAxiom;
5  BEGIN
6    Append(GeneratedCode, GenDeclareSync(BehAxioms));
7    IF Card(BehAxioms)>1 THEN
8      Append(GeneratedCode, GenChooseAxiom(BehAxioms));
9      FOR EACH Axiom ∈ BehAxioms LOOP
10       Append(GeneratedCode,
                GenIfThenEndif(GenAxiomTest(Axiom), GenBehAxiom(Axiom)));
11     END LOOP;
12   ELSE
13     Append(GeneratedCode, GenBehAxiom(First(BehAxioms)));
14   END IF;
15   RETURN GeneratedCode;
16 END GenBehAxiomList;
```

Function GenBehAxiom

This function generates the code corresponding to the subparts of a single axiom.

```
1  FUNCTION GenBehAxiom(Axiom: IN BehAxiom)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4    BoundVars : List_of_Variables := {};
5    RemainingConditions : List_of_Equations := Conditions(Axiom);
6  BEGIN
7    Append(GeneratedCode, GenMatch(Axiom,BoundVars,RemainingConditions));
8    Append(GeneratedCode, GenPre(Axiom,BoundVars,RemainingConditions));
9    Append(GeneratedCode, GenSync(Axiom,BoundVars,RemainingConditions));
10   Append(GeneratedCode, GenPost(Axiom,BoundVars);
11   IF VarsOut(DefinedMethod(Axiom)) ⊄ BoundVars THEN
12        error("Unassigned variables in formal OUT parameters");
13   END IF;
14   IF RemainingConditions /= {} THEN
15        error("Unassigned variables in global condition");
16   END IF;
17   RETURN GeneratedCode;
18 END GenBehAxiom;
```

Function GenMatch

The role of `GenMatch` is to compile the pattern-matching required for the formal `IN` parameters of a method. If the given event is a transition, then `GenMatch` will do nothing.

```
1  FUNCTION GenMatch  (Axiom: IN BehAxiom;
                       BoundVars: IN OUT List_of_Variables;
                       Conditions: IN OUT List_of_Equations)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4    InParam : Term;
5    Var : Variable;
6  BEGIN
7    FOR EACH InParam ∈ TermsIn(DefinedMethod(Axiom)) LOOP
8      IF NOT Linear(InParam) OR NOT NormalForm(InParam) THEN
9        error("Illegal formal IN parameter");
10     END IF;
11     IF NOT IsSimpleVar(InParam) THEN
12       CreatePatternMatchingFunction(InParam);
13       Append(GeneratedCode,GenCallPatternMatchingFunction(InParam));
14       Append(GeneratedCode,GenTestMatchResult);
15     END IF;
16     FOR EACH Var ∈ Vars(InParam) LOOP
17       IF Var ∈ BoundVars THEN
18         Append(GeneratedCode,GenEqualityTest(Var,Renaming(Var)));
19       ELSE
20         Append(BoundVars,Var);
21       END IF;
22     END LOOP;
23     Append(GeneratedCode,GenCalculableConditions(Conditions,BoundVars));
24   END LOOP;
25   RETURN GeneratedCode;
26 END GenMatch;
```

Function GenPre

This function generates the code for all preconditions of the given axiom.

```
1  FUNCTION GenPre(Axiom: IN BehAxiom;
                   BoundVars: IN OUT List_of_Variables;
                   Conditions: IN OUT List_of_Equations)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4    Pre : Term;
5    Var : Variable;
6  BEGIN
7    FOR EACH Pre ∈ Preconditions(Axiom) LOOP
8      IF NOT Linear(Pre) OR NOT NormalForm(Pre) THEN
9        error("Illegal precondition");
10     END IF;
11     Append(GeneratedCode,GenGet(Pre));
12     IF NOT IsSimpleVar(Pre) THEN
13       CreatePatternMatchingFunction(Pre);
14       Append(GeneratedCode,GenCallPatternMatchingFunction(Pre));
```

```
15        Append(GeneratedCode,GenTestMatchResult);
16     END IF;
17     FOR EACH Var ∈ Vars(Pre) LOOP
18        IF Var ∈ BoundVars THEN
19           Append(GeneratedCode,GenEqualityTest(Var,Renaming(Var)));
20        ELSE
21           Append(BoundVars,Var);
22        END IF;
23     END LOOP;
24     Append(GeneratedCode,GenCalculableConditions(Conditions,BoundVars));
25   END LOOP;
26   RETURN GeneratedCode;
27 END GenPre;
```

Function GenPost

`GenPost` generates the code for all postconditions of the given axiom.

```
1  FUNCTION GenPost(Axiom: IN BehAxiom;
                    BoundVars: IN List_of_Variables)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4    Post : Term;
5  BEGIN
6    FOR EACH Post ∈ Postconditions(Axiom) LOOP
7      IF Vars(Post) ⊄ BoundVars THEN
8        error("Unassigned variable in postcondition");
9      END IF;
10     Append(GeneratedCode,GenPut(Post));
11   END LOOP;
12   RETURN GeneratedCode;
13 END GenPost;
```

Function GenCalculableConditions

This function generates code for all calculable equations of the given global conditions. *Calculable* means that the necessary variables have been assigned. The equations for which code has been produced are removed from the given `Conditions` list to avoid generating multiple identical tests.

```
1  FUNCTION GenCalculableConditions(Conditions: IN OUT List_of_Equations;
                                    BoundVars: IN List_of_Variables)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4    Eq : Equation;
5  BEGIN
6    FOR EACH Eq ∈ Conditions LOOP
7      IF Vars(Eq) ⊂ BoundVars THEN
8        Append(GeneratedCode,GenTestEquation(Eq));
9        Remove(Eq,Conditions);
10     END IF;
11   END LOOP;
```

```
12   RETURN GeneratedCode;
13 END GenCalculableConditions;
```

Function GenSync

`GenSync` produces the code for the synchronization part of the given axiom. The synchronization expression is a binary tree which we tilt to the right in order to facilitate the code generation. For instance, $(e_1..e_2)..e_3$ is transformed into $e_1..(e_2..e_3)$. This operation is purely syntactical and does not change the semantics of the original expression; it just makes the structure of the generated code more regular.

```
1   FUNCTION GenSync(Axiom: IN BehAxiom;
                     BoundVars: IN OUT List_of_Variables;
                     Conditions: IN OUT List_of_Equations)
2   RETURN InstructionList IS
3     GeneratedCode : InstructionList := {};
4     CurrSync : Event := TiltRightwards(Synchronization(Axiom));
5   BEGIN
6     Append(GeneratedCode,GenSendFirst(CurrSync));
7     Append(GeneratedCode,GenRepeat);
8     Append(GeneratedCode,GenWaitNextEvent);
9     Append(GeneratedCode,GenSelectSync(CurrSync,BoundVars,Conditions));
10    Append(GeneratedCode,GenUntilDone(CurrSync));
11    Append(GeneratedCode,GenCalculableConditions(Conditions,BoundVars));
12    RETURN GeneratedCode;
13  END GenSync;
```

Function GenSendFirst

The purpose of this function is to generate `Send` instructions for a maximal amount of method calls so that all simultaneous events are really fired in parallel. For instance, for the synchronization $(m_1..m_2)\&((m_3\&m_4)..m_5)$, the `FirstInvocations` are $\{m_1,m_3,m_4\}$ and these should be started in parallel by sending the requests at the same moment.

```
1   FUNCTION GenSendFirst(CurrSync: IN Event;
                          BoundVars: IN OUT List_of_Variables)
2   RETURN InstructionList IS
3     GeneratedCode : InstructionList := {};
4     e : Event;
5   BEGIN
6     FOR EACH e ∈ FirstInvocations(CurrSync) LOOP
7       Append(GeneratedCode,GenSend(e,BoundVars));
8     END LOOP;
9     RETURN GeneratedCode;
10  END GenSendFirst;
```

Function GenSend

`GenSend` simply generates `Send` instructions for the given method invocation.

```
1  FUNCTION GenSend(Method: IN Event;
                    BoundVars: IN OUT List_of_Variables)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4    Var : Variable;
5  BEGIN
6    IF Vars(TermsIn(Method)) ⊄ BoundVars THEN
7      error("Unassigned actual IN parameter");
8    END IF;
9    Append(GeneratedCode,Instruction(Send,GetXid(Method),TermsIn(Method)));
10   RETURN GeneratedCode;
11 END GenSend;
```

Function GenReceive

`GenReceive` generates a `Receive` instruction followed by all the code necessary for assigning variables and testing the implicit conditions.

```
1  FUNCTION GenReceive(Method: IN Event;
                       BoundVars: IN OUT List_of_Variables;
                       Conditions: IN OUT List_of_Equations)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4    OutParam : Term;
5    Var : Variable;
6  BEGIN
7    Append(GeneratedCode,Instruction(Receive,GetXid(Method),TermsOut(Method)));
8    FOR EACH OutParam ∈ TermsOut(Method) LOOP
9      IF NOT Linear(OutParam) OR NOT NormalForm(OutParam) THEN
10       error("Illegal actual OUT parameter");
11     END IF;
12     IF NOT IsSimpleVar(OutParam) THEN
13        CreatePatternMatchingFunction(OutParam);
14        Append(GeneratedCode,GenCallPatternMatchingFunction(OutParam));
15        Append(GeneratedCode,GenTestMatchResult);
16     END IF;
17     FOR EACH Var ∈ Vars(OutParam) LOOP
18        IF Var ∈ BoundVars THEN
19           Append(GeneratedCode,GenEqualityTest(Var,Renaming(Var)));
20        ELSE
21           Append(BoundVars,Var);
22        END IF;
23     END LOOP;
24     Append(GeneratedCode,GenCalculableConditions(Conditions,BoundVars));
25   END LOOP;
26   RETURN GeneratedCode;
27 END GenReceive;
```

Function GenSelectSync

`GenSelectSync` inspects the given synchronization expression in order to generate the control structure leading to the adequate actions in the event loop.

```
1  FUNCTION GenSelectSync(CurrSync: IN Event;
                          BoundVars: IN OUT List_of_Variables;
                          Conditions: IN OUT List_of_Equations)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4  BEGIN
5    CASE SyncKind(CurrSync) IS
6      WHEN seq THEN
7        Append(GeneratedCode,GenSelectSeq(CurrSync,BoundVars,Conditions));
8      WHEN sim THEN
9        Append(GeneratedCode,GenSelectSim(CurrSync,BoundVars,Conditions));
10     WHEN alt THEN
11       Append(GeneratedCode,GenSelectAlt(CurrSync,BoundVars,Conditions));
12     WHEN basic THEN
13       Append(GeneratedCode,GenSelectBasic(CurrSync,BoundVars,Conditions));
14   END CASE;
15   RETURN GeneratedCode;
16 END GenSelectSync;
```

Function GenSelectSeq

This function generates the control structure necessary for managing sequential synchronizations. The call `GenIfGTE(m)` where m is the set $\{m_1,...,m_n\}$ produces the code "`if CurrEvent>=Reply(`m_1`) or ... or CurrEvent>=Reply(`m_n`) then`". The call `GenIfDone(e)` with $e=m_1 \& m_2$ results in the code "`if Done(m1) and Done(m2) then`" and if $e=m_1+m_2$ then it returns "`if Done(m1) or Done(m2) then`".

```
1  FUNCTION GenSelectSeq(CurrSync: IN Event;
                         BoundVars: IN OUT List_of_Variables;
                         Conditions: IN OUT List_of_Equations)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4  BEGIN
5    Append(GeneratedCode,GenIfGTE(FirstInvocations(Left(CurrSync))));
6    Append(GeneratedCode,GenSelectSync(Left(CurrSync),BoundVars,Conditions));
7    IF SyncKind(Left(CurrSync)) ∈ {sim,alt} THEN
8      Append(GeneratedCode,GenIfDone(Left(CurrSync)));
9      Append(GeneratedCode,GenSendFirst(Right(CurrSync)));
10     Append(GeneratedCode,GenEndIf);
11   ELSE
12     Append(GeneratedCode,GenSendFirst(Right(CurrSync)));
13   END IF;
14   Append(GeneratedCode,GenElse);
15   Append(GeneratedCode,GenIfGTE(FirstInvocations(Right(CurrSync))));
16   Append(GeneratedCode,GenSelectSync(Right(CurrSync),BoundVars,Conditions));
17   Append(GeneratedCode,GenEndIf);
18   RETURN GeneratedCode;
19 END GenSelectSeq;
```

Function GenSelectSim

This function generates the control structure necessary for managing simultaneous synchronizations.

```
1  FUNCTION GenSelectSim(CurrSync: IN Event;
                         BoundVars: IN OUT List_of_Variables;
                         Conditions: IN OUT List_of_Equations)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4    LeftBoundVars : List_of_Variables := Copy(BoundVars);
5    RightBoundVars : List_of_Variables := Copy(BoundVars);
6  BEGIN
7    Append(GeneratedCode,GenIfGTE(FirstInvocations(Left(CurrSync))));
8    Append(GeneratedCode,GenSelectSync(Left(CurrSync),LeftBoundVars,Conditions));
9    Append(GeneratedCode,GenElse);
10   Append(GeneratedCode,GenIfGTE(FirstInvocations(Right(CurrSync))));
11   Append(GeneratedCode,GenSelectSync(Right(CurrSync),RightBoundVars,Conditions));
12   Append(GeneratedCode,GenEndIf);
13   BoundVars := LeftBoundVars ∪ RightBoundVars;
14   RETURN GeneratedCode;
15 END GenSelectSim;
```

Function GenSelectAlt

This function generates code for exploring sequentially all alternatives of a synchronization.

```
1  FUNCTION GenSelectAlt(CurrSync: IN Event;
                         BoundVars: IN OUT List_of_Variables;
                         Conditions: IN OUT List_of_Equations)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4    LeftBoundVars : List_of_Variables := Copy(BoundVars);
5    RightBoundVars : List_of_Variables := Copy(BoundVars);
6  BEGIN
7    Append(GeneratedCode,GenIfGTE(FirstInvocations(Left(CurrSync))));
8    Append(GeneratedCode,GenIfFailureSendNext(Left(CurrSync),Right(CurrSync)));
9    Append(GeneratedCode,GenSelectSync(Left(CurrSync),LeftBoundVars,Conditions));
10   Append(GeneratedCode,GenElse);
11   Append(GeneratedCode,GenIfGTE(FirstInvocations(Right(CurrSync))));
12   IF SyncKind(Right(CurrSync))=alt THEN
13     Append(GeneratedCode,
              GenIfFailureSendNext(Left(Right(CurrSync)),Right(Right(CurrSync))));
14   END IF;
15   Append(GeneratedCode,GenSelectSync(Right(CurrSync),RightBoundVars,Conditions));
16   Append(GeneratedCode,GenEndIf);
17   BoundVars := LeftBoundVars ∩ RightBoundVars;
18   RETURN GeneratedCode;
19 END GenSelectAlt;
```

Function GenSelectBasic

`GenSelectBasic` generates the code for receiving the reply to a successful method call.

```
1  FUNCTION GenSelectBasic(CurrSync: IN Event;
                           BoundVars: IN OUT List_of_Variables;
                           Conditions: IN OUT List_of_Equations)
2  RETURN InstructionList IS
3    GeneratedCode : InstructionList := {};
4  BEGIN
5    Append(GeneratedCode,GenIfSuccess(CurrSync));
6    Append(GeneratedCode,GenReceive(CurrSync,BoundVars,Conditions));
7    Append(GeneratedCode,GenEndIf);
8    RETURN GeneratedCode;
9  END GenSelectBasic;
```

Appendix C.4
Summary of Restrictions to CO-OPN Objects

Let us briefly review the list of restrictions to the Object part of CO-OPN. There are mainly four kinds of limitations, all of which are detectable by static analysis of the source text:

1. We have the restriction that the algebraic terms upon which pattern-matching is performed must be linear and in normal form, as in the left-hand sides of conditional equations in Adt modules of CO-OPN (definition 44 on page 240).
2. There are limitations on the sharing of variables between the branches of a simultaneity: Variables cannot be assigned by two independent threads, and variables cannot be used to transmit information between independent threads. See definition 44 on page 240 for a complete description.
3. There are limitations on the behavioural axioms:
   - There must be no “recursive” call from a transition, i.e. during the stabilization of an Object *o*, no call to a method defined in *o* is acceptable since unstable Objects are not allowed to serve invocations.
   - There must be no transition without any precondition or synchronization, otherwise it will always be firable, thus violating the hypothesis that all stabilizations are finite.
4. There is a limitation, seen in section 6.4.3, on the inter-Object structure and numbering which excludes all cases where the branches of a simultaneous synchronization do not have the same priorities when viewed from the emitter and from a shared Object. This requires a static analysis of all synchronization combinations to be performed.

Appendix D.
The Collaborative Diary Specification

In this specification all Objects except `Network` should be duplicated for each of the three users of the collaborative diary. This can be done either manually or by transforming them into generic Objects (which we did not do for the sake of legibility). In appendix E an example is given where the instance of Object `DAL` for user number one is called `DAL1`, and so on.

```
(:-------------------------------------------------------------------------------------------------*
| Specification of ADT Time: Sorts 'date' and 'daytime' with their associated comparison operators.  |
*-------------------------------------------------------------------------------------------------:)
ADT Time;
INTERFACE
  USE Naturals, Booleans;
  SORTS date, daytime;
  GENERATORS
    DMY _ _ _ : natural natural natural -> date;
    HM _ _    : natural natural -> daytime;
  OPERATIONS
    _ = _ : date date -> boolean;
    _ = _ : daytime daytime -> boolean;
    _ < _ : daytime daytime -> boolean;
BODY
  AXIOMS
    ((DMY day1, month1, year1) = (DMY day2, month2, year2)) =
      (day1=day2) AND (month1=month2) AND (year1=year2);

    ((HM hour1, minute1) = (HM hour2, minute2)) =
      (hour1=hour2) AND (minute1=minute2);

    ((HM hour1, minute1) < (HM hour2, minute2)) =
      (hour1<hour2) OR (hour1=hour2 AND minute1<minute2);
  WHERE
    day1,day2,month1,month2,year1,year2 : natural;
    hour1,hour2,minute1,minute2 : natural;
END Time;
```

```
(:-------------------------------------------------------------------------------------------------*
| Specification of ADT Event.                                                                      |
| An event is an entry in the diary consisting of a date, a beginning and an end time within normal |
| working hours, as well as a string describing in natural language the nature of the event.       |
*-------------------------------------------------------------------------------------------------:)
ADT Event;
INTERFACE
  USE Booleans, String, Time;
```

```
  SORTS event;
  GENERATORS
    < _ _ _ _ > : date, daytime, daytime, string -> event;
  OPERATIONS
    _ = _ : event event -> boolean;
    Overlapping _ _ : event event -> boolean;
BODY
  AXIOMS
    (: Define equality :)
    (< day1, start1, end1, cmnt1 > = < day2, start2, end2, cmnt2 >) =
      (day1=day2) AND (start1=start2) AND (end1=end2) AND (cmnt1=cmnt2);

    (: Define 'Overlapping' working hours :)
    Overlapping    < day1, start1, end1, cmnt1 >
                   < day2, start2, end2, cmnt2 > =
      (day1=day2) AND (start1 <= end2) AND (end1 > start2);
  WHERE
    day1, day2 : date;
    start1, start2 : daytime;
    end1, end2 : daytime;
    cmnt1, cmnt2 : string;
END Event;
```

```
(:-------------------------------------------------------------------------------------------------------*
| Specification of ADT ListEvent.                                                                       |
| ListEvent is the unordered list of all events composing a diary. Operation '-' is partial.           |
*-------------------------------------------------------------------------------------------------------:)
ADT ListEvent;
INTERFACE
  USE Booleans, Event;
  SORTS listevent;
  GENERATORS
    []    : -> listevent;
    _ + _ : event listevent -> listevent;
  OPERATIONS
    _ + _ : listevent listevent -> listevent;
    _ - _ : listevent event -> listevent;
    _ isin _ : event listevent -> boolean;
    _ = _ : listevent listevent -> boolean;
BODY
  AXIOMS
    ([] + l1) = l1;
    ((e + l1) + l2) = (e + (l1 + l2));

    (e1=e2) = true => ((e1+l1) - e2) = l1;
    (e1=e2) = false => ((e1+l1) - e2) = (l1 - e2) + e1;

    e1 isin [] = false;
    (e1 isin (l1+e2)) = (e1 = e2) or (e1 isin l1);

    ([] = []) = true;
    ((e+l1) = []) = false;
    ([] = (e+l1)) = false;
    ((e1+l1) = (e2+l2)) = (e1=e2) and (l1=l2);
  WHERE
    l1, l2 : listevent;
    e,e1,e2 : event;
END ListEvent;
```

```
(:-------------------------------------------------------------------------------------------------------*
| Specification of ADT Action.                                                                          |
| An action is a reified view of one of the operations which can be applied to the diary. The purpose   |
| is to be able to validate an operation on the network before applying it definitively to the diary.   |
*-------------------------------------------------------------------------------------------------------:)
ADT Action;
INTERFACE
  USE  Event, ListEvent, Booleans;
  SORTS action;
  GENERATORS
    AddEvent _ : event -> action;
    Update _ _ : event, event -> action;
    Cancel _   : event -> action;
  OPERATIONS
    _ = _ : action action -> boolean;
    Conflicting _ _ : action action -> boolean;
    Consistent _ _  : action listevent -> boolean;
BODY
  SORT
    actionKind; (: for testing equality of actions :)
  GENERATORS
    AddEventKind,UpdateKind,CancelKind : -> actionKind;
  OPERATIONS
    Kind _ : action -> actionKind; (: auxiliary operation for defining '=' :)
    GetFirstEvent _ : action -> event; (: for checking for conflicts :)
    GetSecondEvent _ : action -> event; (: idem :)
  AXIOMS
    Kind(AddEvent e1) = AddEventKind;
    Kind(Update(e1,e2)) = UpdateKind;
    Kind(Cancel e1) = CancelKind;

    (AddEvent e1 = Addevent e2) = (e1=e2);
    (Update e1,e2 = Update e3,e4) = (e1=e3) and (e2=e4);
    (Cancel e1 = Cancel e2) = (e1=e2);
    !(Kind(a1)=Kind(a2)) => (a1=a2) = false;

    GetFirstEvent(AddEvent e1) = e1;
    GetFirstEvent(Update e1,e2) = e1;
    GetFirstEvent(Cancel e1) = e1;
    GetSecondEvent(AddEvent e1) = e1;
    GetSecondEvent(Update e1,e2) = e2;
    GetSecondEvent(Cancel e1) = e1;

    Conflicting (AddEvent e1, Addevent e2) = not(e1=e2) and Overlapping (e1,e2);
    Conflicting (Update(e1,e2), Update(e3,e4)) =
       (not(e1=e3) and not(e2=e4)) and (Overlapping (e1,e3) or Overlapping (e2,e4));
    Conflicting (Cancel e1, Cancel e2) = not(e1=e2) and Overlapping (e1,e2);
    !(Kind(a1)=Kind(a2)) =>
       Conflicting (a1,a2) = Overlapping (GetFirstEvent(a1),GetFirstEvent(a2)) or
             Overlapping (GetSecondEvent(a1),GetSecondEvent(a2));

    Consistent(AddEvent e1,[]) = true;
    Consistent(AddEvent e1, e2+le) =
       (not Overlapping(e1,e2)) and Consistent(AddEvent e1, le);
    Consistent(Update(e1,e2), le) =
       Consistent(Cancel e1, le) and Consistent(AddEvent e2, le);
    Consistent(Cancel e1, le) = e1 isin le;

  WHERE
```

```
   a,a1,a2 : action;
   e1,e2 : event;
   le : listevent;
END Action;


(:-------------------------------------------------------------------------------------------------------*
| Specification of Object ADR (Abstract Document Representation).                                        |
| Object ADR encapsulates the low-level data structures for the replicated diary.                        |
*-------------------------------------------------------------------------------------------------------:)
OBJECT ADR;
INTERFACE
  USE Booleans, ListEvent, Event;
  METHODS
    Consult _ : listevent;
    AddEvent _ : event;
    Update _ _ : event, event;
    Cancel _ : event;
BODY
  PLACES
    diary: listevent;
  INITIAL
    diary [];
  AXIOMS
    Consult(le) : diary le -> diary le;
    AddEvent(e) : diary le -> diary(e + le);
    (e1 isin le) = true =>
      Update(e1, e2): diary le ->diary (e2 + (le - e1));
    (e isin le) = true =>
      Cancel e : diary le -> diary (le - e);
  WHERE
    e, e1, e2 : event;
    le : listevent;
END ADR;


(:-------------------------------------------------------------------------------------------------------*
| Specification of Object DAL (Data Access Layer).                                                       |
| Object DAL filters all accesses to the ADR part of a diary.                                            |
*-------------------------------------------------------------------------------------------------------:)
OBJECT DAL;
INTERFACE
  USE Booleans, ADR, ListEvent, Event, Action;
  METHODS
    Act _                : action;
    DisplayModif _       : listevent;
    DisplayConflict _    : action;
    Transmit _           : action;
    Confirm _            : action;
    Conflict _           : action;

BODY
  PLACES
    confirmed: listevent;
    conflicts: action;
    wait-transmit: action;
  AXIOMS

    Consistent(a,l) = true =>
      Act a WITH Consult(l) : -> wait-transmit a;
```

```
   Consistent(a,l) = false =>
     Act a WITH Consult(l) : -> conflicts a;

   DisplayModif l : confirmed l ->;

   DisplayConflict a : conflicts a ->;

   Transmit a : wait-transmit a ->;

   Confirm (AddEvent(e)) WITH AddEvent(e) .. Consult(l) : -> confirmed l;

   Confirm (Update(e1, e2))
     WITH Update(e1, e2) .. Consult(l)
     : -> confirmed l;

   Confirm (Cancel e)
     WITH Cancel e .. Consult(l)
     : -> confirmed l;

   Conflict a
     : -> conflicts a;

  WHERE
    e, e1, e2 : event;
    a : action;
    l : listevent;

END DAL;


(:-------------------------------------------------------------------------------------------------------*
| Specification of ADT Vote.                                                                            |
| ADT Vote simply defines an enumerated type (VoteCommit,VoteAbort) for which operation 'Result' is     |
| similar to a boolean conjunction.                                                                     |
*-------------------------------------------------------------------------------------------------------:)
ADT Vote;
INTERFACE
  USE Booleans;
  SORTS vote;
  GENERATORS
    VoteCommit : -> vote;
    VoteAbort : -> vote;
  OPERATIONS
    Result _ _ : vote vote -> vote;
    VoteForCondition _ : boolean -> vote;
BODY
  AXIOMS
    Result VoteCommit VoteCommit = VoteCommit;
    Result VoteCommit VoteAbort = VoteAbort;
    Result VoteAbort VoteCommit = VoteAbort;
    Result VoteAbort VoteAbort = VoteAbort;

    VoteForCondition true = VoteCommit;
    VoteForCondition false = VoteAbort;
END Vote;


(:-------------------------------------------------------------------------------------------------------*
| Specification of ADT Decision.                                                                        |
| ADT Decision defines an enumerated type (DecideCommit,DecideAbort) for which the only operation is    |
| to convert a 'vote' into a 'decision'.                                                                |
*-------------------------------------------------------------------------------------------------------:)
```

```
ADT Decision;
INTERFACE
  USE Vote;
  SORTS decision;
  GENERATORS
    DecideCommit : -> decision;
    DecideAbort : -> decision;
  OPERATIONS
    DecisionForVote _ : vote -> decision;
BODY
  AXIOMS
    DecisionForVote VoteCommit = DecideCommit;
    DecisionForVote VoteAbort = DecideAbort;
END Decision;
```

```
(:-----------------------------------------------------------------------------------------------------*
| Specification of ADT Message.                                                                        |
| ADT Message defines a variant record the role of which is to transform the given data into a message. |
*-----------------------------------------------------------------------------------------------------:)
ADT Message;
INTERFACE
  USE Action, Vote, Decision;
  SORTS message;
  GENERATORS
    Pack _ : action -> message;
    Pack _ : vote -> message;
    Pack _ : decision -> message;
BODY
  (: No axioms :)
END Message;
```

```
(:-----------------------------------------------------------------------------------------------------*
| Specification of ADT ID.                                                                             |
| ADT ID defines an enumerated type of three identifiers to be used as network addresses for the diary  |
| users. This is a static solution which simplifies the specification.                                 |
*-----------------------------------------------------------------------------------------------------:)
ADT ID;
INTERFACE
  USE Booleans;
  SORTS id;
  GENERATORS
    id1, id2, id3 : -> id;
  OPERATIONS
    _ = _ : id id -> boolean;
BODY
    ;; Syntactic equality is fine for such a simple type:
    (ida = idb) => (ida = idb) = true;
    !(ida = idb) => (ida = idb) = false;
  WHERE
    ida,idb : id;
END ID;
```

```
(:-----------------------------------------------------------------------------------------------------*
| Specification of Object Network.                                                                     |
| This is a centralized view of the services expected from the real communication medium. This Object   |
| will have to be unfolded at the moment of implementation so as to provide each diary user with its   |
| own refined Network Object which sends and receives requests intelligently instead of broadcasting   |
| and filtering like here.                                                                             |
*-----------------------------------------------------------------------------------------------------:)
```

```
OBJECT Network;
INTERFACE
  USE Message, ID;
  METHODS
    Put _ _ _ : message id id;
    Get _ _ _ : message id id;
BODY
  PLACES
    Channel _ _ _ : message id id;
  AXIOMS
    Put msg iddest idorigin : -> Channel msg iddest idorigin;
    Get msg iddest idorigin : Channel msg iddest idorigin ->;
  WHERE
    msg : message;
    iddest, idorigin : id;
END Network;
```

```
(:-------------------------------------------------------------------------------------------------*
| Specification of Object DSA (Distributed Synchronization Algorithm).                             |
| The role of the DSA is to ensure that the information in all replicated diaries is consistent. To |
| this end, it implements an atomic commitment protocol known as the "two phase commit". At any moment |
| each diary user has at most one action to validate; if another user wants to validate his own action |
| exactly at the same time, then it must be controlled that these two actions are compatible.      |
| To simplify the specification, we have a static configuration with three permanent diary users.  |
*-------------------------------------------------------------------------------------------------:)
OBJECT DSA;
INTERFACE
  USE
    Booleans,
    Action, Vote, Decision, Message, ID,
    DAL, Network;
BODY
  TRANSITIONS
    DispatchRequest,
    ReceiveRequest,
    CollectVote,
    DispatchDecision,
    ReceiveDecision;
  PLACES
    MyIdHolder _ : id;
    MyDecision _ : decision;
    PendingActions _ _ : id action;
    OwnAction _ : boolean;
  INITIAL
    OwnAction false;
    MyIdHolder id1;   ;; for instance, or id2 or id3.
  AXIOMS
;; My DAL wants me to submit an action for approval:
    DispatchRequest
      WITH Transmit a ..
           (Put (Pack a) id1 myId ..
           (Put (Pack a) id2 myId ..
            Put (Pack a) id3 myId))
           : OwnAction false, MyIdHolder myId
           -> OwnAction true, PendingActions myId a, MyIdHolder myId;

;; When all participants have voted, put the decision in MyDecision for DispatchDecision:
    CollectVote
      WITH Get (Pack v1) myId id1 &
           (Get (Pack v2) myId id2 &
```

```
          Get (Pack v3) myId id3)
         : PendingActions myId a, MyIdHolder myId
         -> MyDecision DecisionForVote(Result(v1,Result(v2,v3))),
            PendingActions myId a, MyIdHolder myId;

;; I have decided whether my action was ok or not, let everybody know about it:
   DispatchDecision
     WITH Put (Pack d) id1 myId ..
          (Put (Pack d) id2 myId ..
           Put (Pack d) id3 myId)
          : MyDecision d, OwnAction true, MyIdHolder myId
          -> OwnAction false, MyIdHolder myId;

;; This message is not from me, and I have some action pending
;; check if they conflict.
   (myId = anyId)=false =>
     ReceiveRequest
       WITH   Get (Pack a) myId anyId ..
              Put (Pack VoteForCondition(Conflicting a myAction)) anyId myId
              : OwnAction true, PendingActions myId myAction, MyIdHolder myId
              -> PendingActions anyId a, PendingActions myId myAction,
                 MyIdHolder myId;

;; This message does not come from me, and I have no own action pending
;; therefore, commit:
   (myId = anyId)=false =>
     ReceiveRequest
       WITH   Get (Pack a) myId anyId ..
              Put (Pack VoteCommit) anyId myId
              : OwnAction false, MyIdHolder myId
              -> PendingActions anyId a, OwnAction false, MyIdHolder myId;

;; This is a message from myself: just commit
   (myId = anyId)=true =>
     ReceiveRequest
       WITH   Get (Pack a) myId anyId ..
              Put (Pack VoteCommit) myId myId
              : MyIdHolder myId
              -> MyIdHolder myId;

;; Receive the positive decision of the action originator 'anyId' (it could be myself)
;; and transfer the action as committed to my DAL:
   ReceiveDecision
     WITH Get (Pack DecideCommit) myId anyId .. Confirm a
          : PendingActions anyId a, MyIdHolder myId
          -> MyIdHolder myId;

;; Receive a negative decision about my action and tell my DAL there is a conflict:
   (myId = anyId)=true =>
     ReceiveDecision
       WITH   Get (Pack DecideAbort) myId anyId .. Conflict a
              : PendingActions anyId a, MyIdHolder myId
              -> MyIdHolder myId;

;; This action was rejected. Since it was not my action, just forget about it.
   (myId = anyId)=false =>
     ReceiveDecision
       WITH   Get (Pack DecideAbort) myId anyId
              : PendingActions anyId a, MyIdHolder myId
              -> MyIdHolder myId;
```

```
  WHERE
    a,myAction : action;
    v,v1,v2,v3 : vote;
    d : decision;
    myId,anyId : id;
END DSA;
```

Appendix E. An Execution Cycle of the Collaborative Diary

The pictures on the following pages represent a successful insertion of a new meeting into the collaborative diary (specified in appendix D). The numbers preceded by the ⊏ sign indicate the number attributed to the associated Object for the establishment of a total order in the dependency graph. This execution only shows successful invocations, which means however that all `stabilize` requests are displayed. In order to keep the figures legible only two of the three specified diary users are depicted.

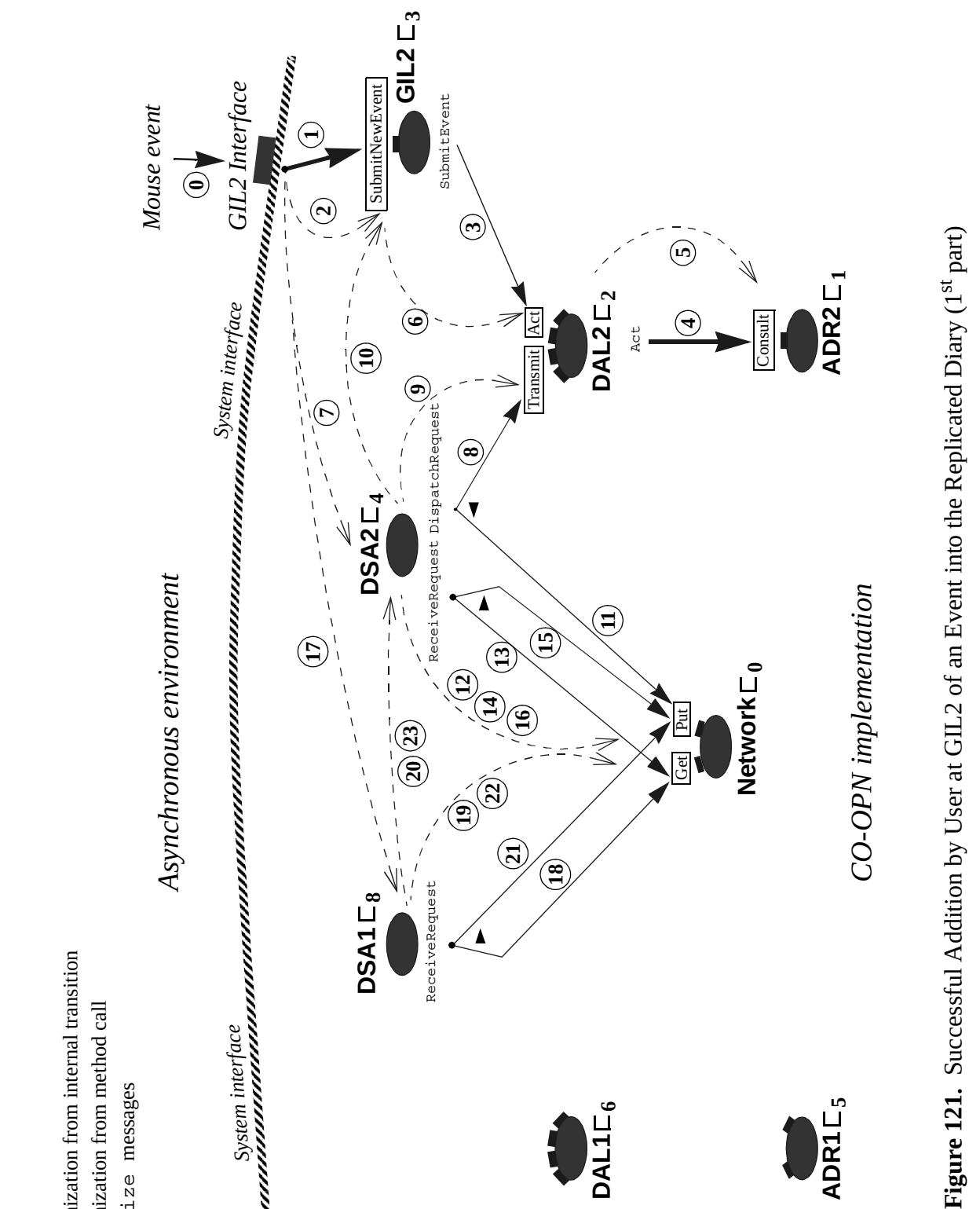


Figure 121. Successful Addition by User at GIL2 of an Event into the Replicated Diary (1st part)

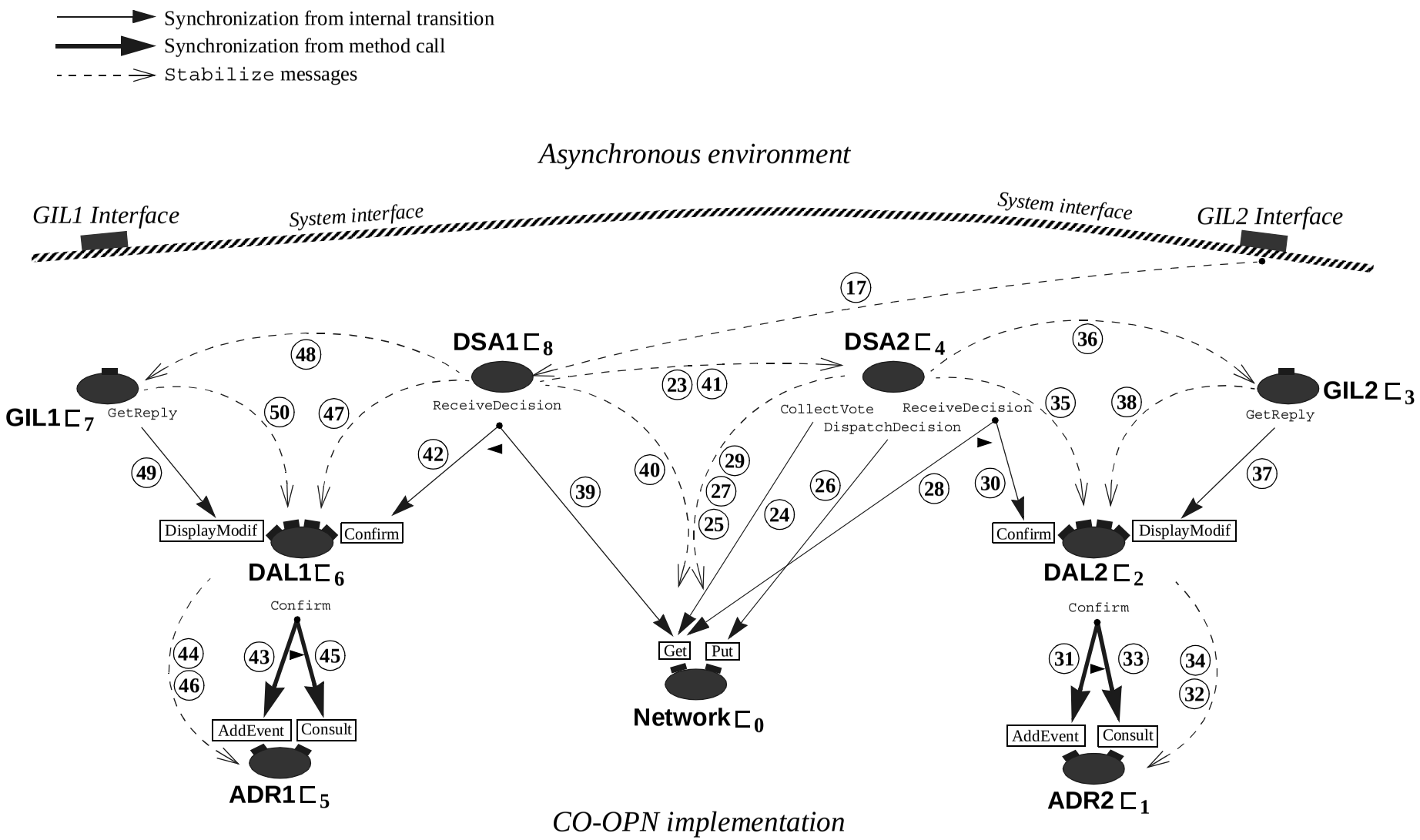


Figure 122. Successful Addition by User at GIL2 of an Event into the Replicated Diary (2nd part)

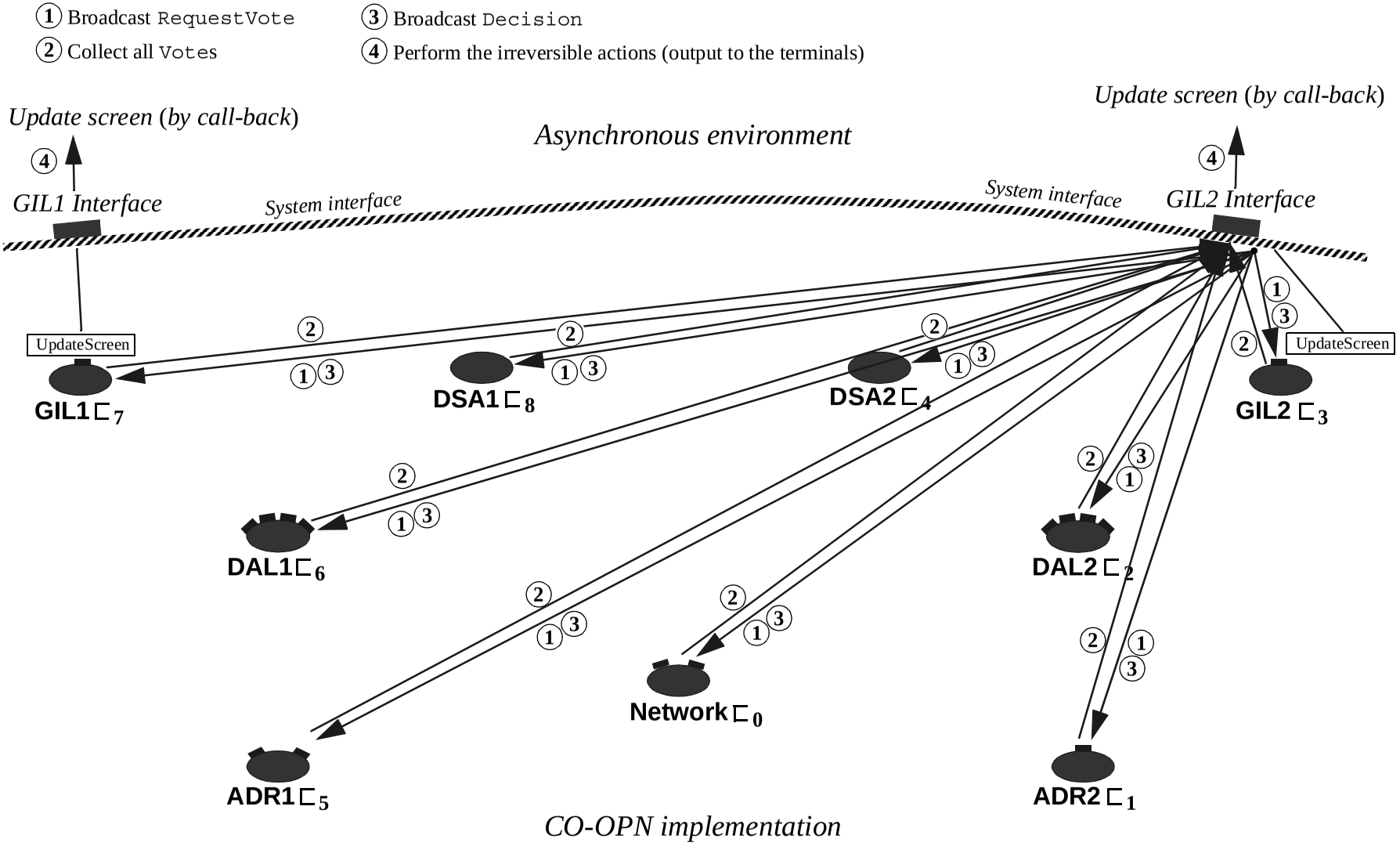


Figure 123. 2PC Atomic Commitment after the Addition by User at GIL2 of an Event into the Replicated Diary

Curriculum Vitae

Geir Jarle Hulaas received a BS and MS in computer science from the Swiss Federal Institute of Technology (EPFL) in Lausanne, Switzerland, in 1991. Until mid-1993 he worked in the same institute as a research assistant on the subject of portable compilation techniques for multi-paradigm programming languages. Then he took part in the design and implementation of a concurrent extension of the C++ language. During three years from April 1994 he was a PhD student at the EPFL working on incremental prototyping techniques for formally specified concurrent systems. In 1995 he received a postgraduate degree in software engineering.

Publications

J. Hulaas, *The Newton Concurrent Object-Oriented System*, DEC-Alpha Innovator Program, Available at Digital Equipment Corporation as CD-ROM AG-Q159A-RE, September 1993.

D. Buchs, J. Hulaas, P. Racloz, M. Buffo, J. Flumet, E. Urland, *SANDS Structured Algebraic Net Development System for CO-OPN*, 16th International Conference on Application and Theory of Petri Nets, Torino, Italy, June 1995, pp. 45-53.

D. Buchs, J. Hulaas, *Incremental Object-Oriented Implementation of Concurrent Systems Based on Prototyping of Formal Specifications*, SIPAR workshop, Biel, Switzerland, Oct. 1995, pp. 141-145.

J. Hulaas, *An Evolutive Distributed Petri Nets Simulator,* 10th European Simulation Multiconference ESM'96, Budapest, Hungary, 2-6 June 1996, pp. 348-352.

D. Buchs, J. Hulaas, *Evolutive Prototyping of Heterogeneous Distributed Systems Using Hierarchical Algebraic Petri Nets,* Procs. IEEE International Conference on Systems, Man and Cybernetics SMC'96, Beijing, China, Oct. 14-17 1996, pp. 3021-3026. Also available as European Esprit Long Term Research Project 20072 "Design for Validation" *(DeVa)* technical report #09, 1996.

D. Buchs, J. Hulaas, P. Racloz, *Exploiting Various Levels of Semantics in CO-OPN for the SANDS Environment Tools,* Tool presentations, 18th Int. Conf. on Application and Theory of Petri Nets, Toulouse, June 23-27, 1997, pp. 34-43.